DATA FRAGMENTATION AND ALLOCATION ALGORITHMS FOR DISTRIBUTED DATABASE DESIGN

By

MINYOUNG RA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1990

"The Lord is my shepherd; I shall not want."
(Psalms 23:1)

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my advisor, Dr. Shamkant B. Navathe, for giving me the opportunity to work in the interesting area of distributed database design and for his continual encouragement and support through the course of this work. I indeed owe him a great deal.

I would like to express my special thanks to my supervisory committee cochair, Dr. Ravi Varadarajan, for his ever-present help and valuable comments on this work. I would like to thank Dr. Sharma Chakravarthy for his useful suggestions and discussions. I would like to thank Dr. Jose C. Principe and Dr. Manuel Bermudez for their time and interest in my research.

I would like to thank Mrs. Sharon Grant for the unceasing help and assistance. I would also like to thank all those unnamed friends who helped me in pursuing my research.

I would like to thank Korea Military Academy and Korean Army for allowing my studying here and for their encouragement and support.

Finally, I would like to thank my family, especially my wife Eunjin and children Shinil and Eunsong, for their continuous prayer and encouragement.

This work was supported by National Science Foundation under grant no. IRI 8716798.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

DATA FRAGMENTATION AND ALLOCATION ALGORITHMS FOR DISTRIBUTED DATABASE DESIGN

By

Minyoung Ra

December 1990

Chairman: Dr. Shamkant B. Navathe
Major Department: Computer and Information Sciences

In a distributed database system data fragmentation and allocation are the major design issues. This dissertation presents a set of algorithms for fragmentation (or partitioning) and allocation. The relational model of data is assumed for convenience. The algorithms can be applied to other data models with minor variations.

The partitioning of a global schema into fragments can be performed in two different ways: vertical partitioning and horizontal partitioning. Vertical partitioning is the process of subdividing the attributes of a relation or a record type into multiple records, thereby creating fragments. Horizontal partitioning is the process that divides a global relation into subsets of tuples, called horizontal fragments. In this dissertation a new vertical partitioning algorithm is presented. This algorithm starts from the attribute affinity matrix by considering it as a complete graph. Then, forming a linearly connected spanning tree, it generates all meaningful fragments simultaneously. A new horizontal partitioning algorithm is also presented, which is developed by applying the same graphical technique as in vertical partitioning.

The need for mixed partitioning arises because database users usually access data subsets which are simultaneously vertical and horizontal fragments of global relations. However, this problem has not been addressed well in the current literature. A mixed partitioning methodology, which first forms a grid by partitioning a global relation vertically and horizontally in an independent fashion and then produces the final fragments (called mixed fragments) by merging the grid cells, is addressed.

In most of the previous allocation work, the unit of allocation is the fragment that results from horizontal partitioning or vertical partitioning. Little work has been done for the allocation of the result of mixed partitioning. This dissertation presents an allocation algorithm for fragments that are generated from our mixed partitioning procedure. In this algorithm a mixed fragment is the unit of allocation. This algorithm is developed based on a heuristic called the pseudoallocation technique.

The contribution of this dissertation consists of providing efficient graph oriented algorithms for partitioning, developing a mixed partitioning approach to distribution and a new approach to fragment allocation in distributed databases.

CHAPTER 1
INTRODUCTION

In recent years, due to the demand for system availability and autonomy, and enabled by advances in database and communication technology, distributed database systems have become an important area of information processing, and it is easy to foresee that their importance will rapidly grow. There are both organizational and technological reasons for this trend; distributed databases eliminate many of the shortcomings of centralized databases and fit more naturally in the decentralized structures of many organizations [Ceri 84].

There are two alternative approaches to the design of data distribution: the top-down and the bottom-up approaches [Cer 83a, Ceri 84]. The former approach is typical of distributed databases developed from scratch, while the latter approach is typical of the development of multi-database systems as the aggregation of existing databases. This dissertation is concerned with the top-down approach.

In the top-down approach, the description of the global schema is given, the fragmentation of the databases is designed and then these fragments are allocated to the sites. The objective of this approach is to evaluate the utility of partitioning data objects into fragments both horizontally and vertically, and then to determine the allocation of fragments to the database sites [Ceri 82]. Optimal and heuristic mathematical models are used to determine the allocation of data to reduce the aggregate cost of transaction processing [Cer 83b, Nava 84].

Figure 1-1 describes an overview of distributed database design when it is performed in a top-down manner [Cer 83b]. The main difference of this overview compared with centralized database design is the distribution design step. Prior to distribution design, a global schema is produced by using standard techniques of centralized database design. During the distribution design step, several local logical schemas for the distributed database are produced. Data objects which belong to the global database schema are fragmented and allocated to the different sites, and associations between objects are taken into account in the process. The design is completed by performing the physical design of the data which are allocated to each site.

1.1 Basic Concepts of Fragmentation and Allocation

The top-down approach to distributed design consists of solving for each global object the design problems of fragmentation and allocation [Cer 83b]. Fragmentation or partitioning is the process of subdividing a global object (entity or relation) into several pieces, called fragments, and allocation is the process of mapping each fragment to one or more sites. Fragments must be appropriate units of allocation. Thus the database designers should define fragments as homogeneous collections of information from the viewpoint of distribution [Ceri 87].

The decomposition of a global schema into fragments can be performed using two different types of fragmentation techniques: horizontal fragmentation and vertical fragmentation. Vertical fragmentation is the process that divides a global object into groups of their attributes, called vertical fragments. In order to be able to access the same data after partitioning, it is required that each fragment include a key attribute of the global object. Horizontal fragmentation is the process that divides a global relation into subsets of tuples, called horizontal fragments. Each fragment is associated with a predicate which indicates the distinguishing property possessed by the instances or tuples of that fragment. Note that fragments need not be disjoint. This means that fragments can have common attributes in vertical fragmentation and common tuples in horizontal fragmentation. Another type of fragmentation is mixed fragmentation. Mixed fragmentation can be built by alternating horizontal and vertical fragmentations.

The allocation of fragments can be either nonredundant or redundant. A nonredundant (nonreplicated) allocation requires each fragment to be mapped to exactly one site, whereas a redundant (replicated) allocation allows each fragment to be mapped to one or more sites. Typically, replication permits higher availability and reliability of the system, though it needs more storage and additional effort for maintaining consistency.

1.2 Proposed Problem

Figure 1-2 shows the Distributed Database Design Tool (D^3T) reference architecture which is used in our research. The term "GRID" is used to refer to the result of applying horizontal partitioning and vertical partitioning algorithms simultaneously to a global relation. It consists of cells called "grid cells." The grid suggests all possible ways in which the global relation in a distributed database may be partitioned. The input information to D^3T is the global schema which consists of a set of relations, together with information about the important transactions on the proposed database. As stated in the previous work [Cer 83b, Nava 84], it is not necessary to collect information on 100% of the expected transactions (that would, of course, be impossible). Since the 80-20 rule applies to most practical situations, it is adequate to supply information regarding the 20% of the heavily used transactions which account for about 80% of the activity against the database. The other input to D^3T shown in Figure 1-2 is the distribution constraints which include preferences or special considerations designers/users may have that would influence partitioning and allocation. In this figure, grid creation is related to vertical fragmentation and horizontal fragmentation, and grid optimization is related to mixed fragmentation. In this dissertation, we will deal with vertical fragmentation, horizontal fragmentation, mixed fragmentation and allocation with no redundancy by presenting new algorithms that have less complexity or provide minimum transaction processing cost.

1.3 Contributions of This Dissertation

In a distributed database system, data fragmentation and allocation are the major design issues. This dissertation presents a set of algorithms for data fragmentation and allocation in a distributed database system environment.

In this dissertation, a number of problems for data fragmentation and allocation are defined and attacked by using graphical techniques. One of the main contributions of this dissertation is the development of the graph oriented partitioning algorithm for vertical and horizontal partitioning. The major feature of this algorithm is that all fragments are generated in one iteration in time $O(n^2)$, where n represents the number of attributes, that is more efficient than the previous approaches. Furthermore, it provides a cleaner alternative without arbitrary objective functions.

Another contribution is the development of the first comprehensive mixed partitioning methodology for this problem.

A graph-based allocation algorithm is also presented along with some heuristics.

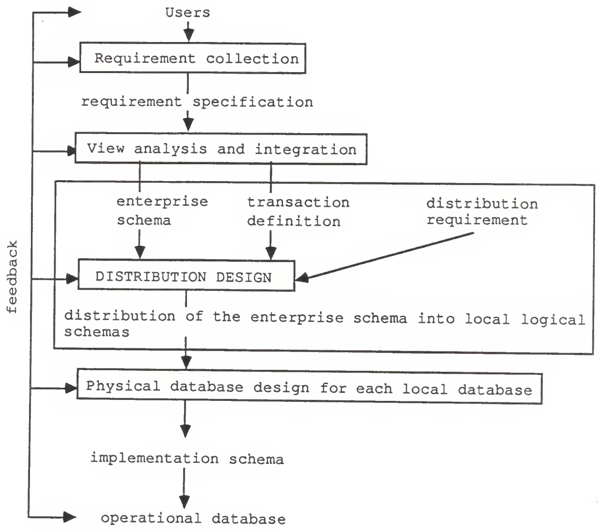

Users
Requirement collection
requirement specification
View analysis and integration
enterprise schema
transaction definition
distribution requirement
DISTRIBUTION DESIGN
distribution of the enterprise schema into local logical schemas
Physical database design for each local database
implementation schema
operational database
feedback

Figure 1-1 The overall distributed database design methodology: Top-down manner [Cer 83b]

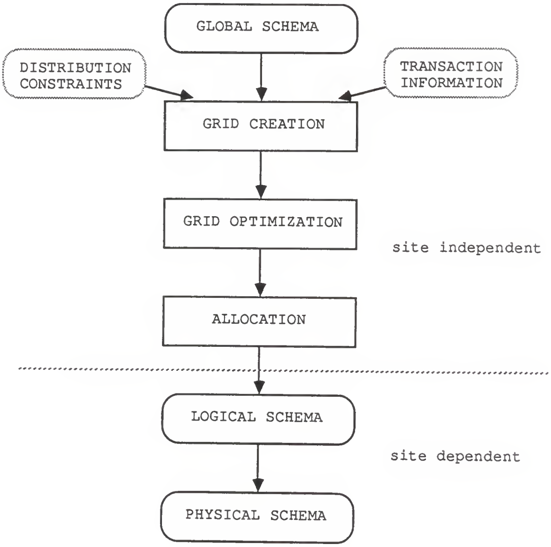


Figure 1-2 Distributed database design tool reference architecture

CHAPTER 2
BACKGROUND AND PROPOSED APPROACH

2.1 Previous Related Work

We review the previous work in the design of data distribution. This review is divided into the work on data fragmentation and the work on data allocation.

2.1.1 Previous Work on Data Fragmentation

Data fragmentation is performed during the design of a database to improve performance of transactions. In order to obtain improved performance, fragments must be "closely matched" to the requirements of the transactions.

The work related to this topic can be classified into the work on vertical partitioning, on horizontal partitioning, and on mixed partitioning.

2.1.1.1 Vertical partitioning

Hoffer and Severance [Hoff 75] measure the affinity between pairs of attributes and try to cluster attributes according to their pairwise affinity by using the bond energy algorithm (BEA) developed in [McCo 72].

Navathe et al. [Nava 84] extend the results of Hoffer and Severance and propose a two-phase approach to vertical partitioning. During the first phase, they use the given input parameters in the form of an attribute usage matrix and transactions to construct the attribute affinity matrix on which clustering is performed. After clustering, iterative binary partitioning is attempted, first with an empirical objective function. The process is continued until no further partitioning results. During the second phase, the fragments can be further refined by incorporating estimated cost factors weighted on the basis of the type of problem being solved.

Cornell and Yu [Corn 87] apply the work of Navathe et al. [Nava 84] to relational databases. They propose an algorithm which decreases the number of disk accesses to obtain an optimal binary partitioning. They show how knowledge of specific physical factors can be incorporated into the overall fragmentation methodology.

Ceri, Pernici and Wiederhold [Ceri 88] extend the work of Navathe et al. [Nava 84] by considering it as a DIVIDE tool and by adding a CONQUER tool. Their CONQUER tool extends the same basic approach in the direction of adding details about operations and physical accesses similar to [Corn 87]. This approach focuses on the decomposition of the design process into several design subproblems and provides no algorithmic improvement in the process of vertical partitioning itself.

2.1.1.2 Horizontal partitioning

Ceri, Negri and Pelagatti [Ceri 82] analyze the horizontal partitioning problem, dealing with the specification of partitioning predicates and the application of horizontal partitioning to different database design problems.

Ceri, Navathe and Wiederhold [Cer 83b] develop an optimization model for horizontal partitioning without replication in the form of a linear integer 0-1 programming problem. They modeled the logical schema of a database as a directed graph with objects as nodes and links as edges, and required the user to specify the information about (1) the data about schema (attribute size, relationship size, cardinality), (2) tabulation of transactions (frequencies, site of origin), and (3) distribution requirement (constraints).

Yu et al. [Yu 85] propose an adaptive algorithm for record clustering, which is conceptually simple and highly intuitive. However, the adaptive approach does not use the transaction information that is useful for partitioning. Hence, it cannot be applied until the execution of transactions.

2.1.1.3 Mixed partitioning

Currently mixed partitioning has not been adequately addressed in the literature. The only work we are aware of is that of Apers [Aper 88], who considers the fragmentation problem together with the allocation problem. In his approach, the fragmentation scheme is the output of the allocation algorithm. Thus we cannot know the fragmentation scheme before allocation.

2.1.2 Previous Work on Data Allocation

Previous work on data allocation has been mainly in two areas: (1) data allocation independent of network parameters, and (2) extension to the pure data allocation problems including the network topology and communication channels. Note that since a distributed file system differs greatly from a distributed database, the solutions for the file allocation problems [Dowd 82] do not characterize solutions to the allocation problems in a distributed database. This is because in the distributed database systems the way the data are accessed is far more complex [Aper 88]. This review concentrates on how the data allocation problem has been attacked in the context of distributed databases and mainly in areas of data allocation itself.

Ceri, Martella and Pelagatti [Cer 80] consider the problem of data allocation for typical database applications with a simple model of transaction execution. However, they do not consider any partitioning.

Ceri, Navathe and Wiederhold [Cer 83b] propose a model for a nonreplicated data allocation in the form of a linear 0-1 integer programming problem. But only horizontal partitioning is considered and a certain query processing strategy is assumed.

Wah [Wah 84] investigates the important problems of file allocation in general. However, he does not consider data fragmentation for database design.

Sacca and Wiederhold [Sacc 85] propose a heuristic allocation algorithm in a cluster of processors in which the user does not address a transaction request to a specific processor. They try to minimize the global cost for allocating data in the given network. However, this approach cannot be applied to the system in which processors are remotely distributed, because, in a remotely distributed system, the user has to request data from a specific site.

Cornell and Yu [Corn 88, Corn 89] propose a strategy to integrate the treatment of relation assignment and query strategy to optimize performance by using the 0-1 integer programming method. They, however, neither consider any partitioning nor join order.

Apers [Aper 88] proposes a model for computing the cost of a completely or partially specified allocation for various cost functions. In this approach fragments are the result of the allocation. So we do not know about the fragmentation scheme before allocation. When fragments are generated, since clustering of fragments is not considered, the generated fragments may contain only a few or even one tuple.

Ceri, Pernici and Wiederhold [Ceri 88] propose a distributed design tool which provides a flexible environment for the design of data distribution. In this paper vertical fragments are allocated to each site according to the DIVIDE and CONQUER tools. They, however, do not consider horizontal partitioning.

2.2 The Outline of the Proposed Approach

This dissertation is primarily concerned with the algorithms for data fragmentation and allocation. We provide an overview of the structure of the dissertation by discussing the basic ideas for solving partitioning and allocation problems in distributed database design.

2.2.1 Vertical Partitioning

In all vertical partitioning algorithms that we have surveyed, the binary partitioning technique has been used for partitioning after clustering attributes. Thus binary partitioning is repeated until all meaningful fragments are determined. It is also necessary that clustering be repeated on two new affinity matrices corresponding to the newly generated fragments. In Chapter 3 we propose a new vertical partitioning algorithm which has less computational complexity and generates all meaningful fragments simultaneously by using a graphical method. This approach is based on the fact that all pairs of attributes in a fragment have high "within fragment affinity" but low "between fragment affinity."

2.2.2 Horizontal Partitioning

In horizontal partitioning, most of the previous approaches that we know of try to select an optimal horizontal partitioning for each relation. In the present approach, however, we do not consider how to select an optimal horizontal partitioning for each relation in the database. We are focusing our attention on identifying all the candidate horizontal partitions by using the same graphical approach as in the vertical partitioning. To this end, we construct a predicate affinity matrix. Based on this matrix we develop a horizontal partitioning algorithm in Chapter 4.

2.2.3.Mixed Partitioning

Mixed partitioning can be accomplished in a sequential manner in one of two ways: first performing vertical partitioning and then horizontally partitioning the vertical partitions (called VH partitioning), or first performing the horizontal partitioning and then vertically partitioning the horizontal partitions (called HV partitioning). Obviously, this is not adequate since it potentially leads to different results and leaves out the possibility of combining fragments at a smaller granularity to produce more efficient data distribution. Thus, in Chapter 5, we propose a uniform mixed partitioning methodology in order to solve the above problems. A grid consisting of "cells" is created by simultaneously applying the vertical partitioning and horizontal partitioning algorithms on the relation. The grid cells will be merged by taking into account the cost factors to minimize the global transaction processing costs. This approach produces optimal mixed fragments that cannot be otherwise produced by independent partitioning models.

2.2.4 Allocation

The problem of data allocation in distributed database systems has been extensively studied. However, in most cases the unit of allocation is the fragment resulting from horizontal partitioning or vertical partitioning. Little work has been done for the allocation of the result of mixed partitioning. So, in Chapter 6, we propose an allocation algorithm for the result of mixed partitioning. A mixed fragment, which comes from our mixed partitioning procedure by considering both vertical partitioning and horizontal partitioning simultaneously, is the unit of allocation. The objective of this algorithm is to allocate the mixed fragments nonredundantly with minimum data transmission cost while satisfying the load balancing constraints. To this end we will present some heuristics that combine fragment allocation and transaction processing strategy. Based on the heuristics we will develop this algorithm

2.2.5 Alternative Ways of Dealing with Fragmentation and Allocation

Distributed database design has been dealt with in terms of fragmentation of data and allocation of these fragments. There may be other alternatives with which these problems can be attacked. In Chapter 7, we describe the various alternatives and evaluate them by comparing one another.

CHAPTER 3
A VERTICAL PARTITIONING ALGORITHM

Vertical partitioning is the process that divides a "global object" which may be a single relation or more like a universal relation into groups of their attributes called vertical fragments [Cer 83a, Nava 84, Corn 87]. It is used during the design of a database to enhance the performance of transactions [Nava 84]. In order to obtain improved performance, fragments must closely match the requirements of the transactions. Vertical partitioning has a variety of applications wherever the "match" between data and transactions can affect performance. That includes partitioning of individual files in centralized environments, data distribution in distributed databases, dividing data among different levels of memory hierarchies, and so on. Figure 3-1 shows the transaction specification for our example that will cover both vertical and horizontal algorithms.

Although this and the subsequent chapters are presented using the relational model, the approach presented in this dissertation is general enough. By replacing the term relation with record or object it can be easily applied to the hierarchical, network or the object-oriented models.

3.1 Overview

The algorithm that we propose starts from the attribute affinity (AA) matrix, which is generated from the attribute usage matrix using the same method as that of the previous approach in [Nava 84]. The attribute usage matrix represents the use of attributes in important transactions. Each row refers to one transaction; the "1" entry in a column indicates that the transaction "uses" the corresponding attributes. Whether the transaction retrieves or updates the relation can also be captured by another column vector with R and U entries for retrieval and update. That information may be used by an empirical objective function as in [Nava 84]. The attribute usage matrix for 10 attributes and 8 transactions is shown in Figure 3-2. Attribute affinity is defined as

$$aff_{ij} = \sum_{k \in \tau} acc_{kij} \quad (i \neq j)$$

where acc_{kij} is the number of accesses of transaction k referencing both attributes i and j. The summation occurs over all transactions that belong to the set of important transactions τ. This definition means that attribute affinity measures the strength of an imaginary bond between the two attributes, predicated on the fact that attributes are used together by transactions. Although affinity aff_{ii} does not have any physical meaning, it is reasonable to define it as follows:

$$aff_{ii} = \sum_{k \in \tau} acc_{ki}$$

where acc_{ki} means the number of accesses of transaction k referencing attribute i. This is reasonable since it shows the "strength" of that attribute in terms of its use in all transactions.

Based on these definitions of attribute affinity, the attribute affinity matrix is defined as follows: It is an n x n matrix for the n-attribute problem whose (i,j) element equals aff_{ij}. Figure 3-3 shows the attribute affinity matrix which was formed from the Figure 3-2.

A note about the attributes: In this proposed technique as well as in the previous techniques, the set of attributes considered may be

(1) the universal set of attributes in the whole database.

(2) the set of attributes in a single relation (or record type or object type).

By using (a), the fragments generated may be interpreted as relations or record types. By using (b), fragments of a single relation are generated.

In previous approaches, a clustering algorithm is applied to the AA matrix. In our present approach, however, we consider the AA matrix as a complete graph called the affinity graph in which an edge value represents the affinity between the two attributes. Then, forming a linearly connected spanning tree, the algorithm generates all meaningful fragments in one iteration by considering a cycle as a fragment. A "linearly connected" tree has only two ends. Figure 3-4 shows the affinity graph corresponding to the AA matrix of Figure 3-3. Note that the AA matrix serves as a data structure for the affinity graph.

The major advantages of the proposed method over that in [Nava 84] are as follows:

(1) There is no need for iterative binary partitioning. The major weakness of iterative binary partitioning is that at each step two new problems are generated increasing the complexity.

(2) The method obviates the need for using any empirical objective functions as in [Nava 84]. As shown in [Corn 87], the intuitive objective functions used in [Nava 84] do not necessarily work well when an actual detailed cost formulation for a specific system is utilized.

(3) The method requires no complementary algorithms such as the SHIFT algorithm of [Nava 84].

(4) The complexity of the algorithm is $O(n^2)$, better than the $O(n^2 \log n)$ complexity of the previous algorithms.

3.2 Definitions and Notations

We shall use the following notation and terminology in the description of our algorithm.

- A,B,C,... denotes nodes.
- a,b,c,... denotes edges.
- p(e) denotes the affinity value of an edge e.
- "primitive cycle" denotes any cycle in the affinity graph.
- "cycle completing edge" denotes a "to be selected" edge that would complete a cycle.
- "cycle node" is that node of the cycle completing edge, which was selected earlier.
- "affinity cycle" denotes a primitive cycle that contains a cycle node. We assume that a cycle means an affinity cycle, unless otherwise stated.
- "former edge" denotes an edge that was selected earlier than the cycle node.
- "cycle edge" is any of the edges forming a cycle.
- "extension of a cycle" refers to a cycle being extended by pivoting at the cycle node.

The above definitions are used in the proposed algorithm to process the affinity graph and to generate possible cycles from the graph. Note that each cycle gives rise to a vertical fragment.

3.3 Fundamental Concepts

Based on the above definitions we would like to explain the mechanism of forming cycles. For example, in Figure 3-5, suppose edges a and b were selected already and c was selected next. At this time, since c forms a primitive cycle, we have to check if it is an affinity cycle. This can be done by checking the possibility of a cycle. "Possibility of a cycle" results from the condition that no former edge exists, or p(former edge) $\leq$ p(all the cycle edges). The primitive cycle a,b,c is an affinity cycle because it has no former edge and satisfies the possibility of a cycle (i.e., the first cycle with three edges is always an affinity cycle). Therefore the primitive cycle a,b,c is marked as a candidate partition and node A becomes a cycle node.

Now let us explain how the extension of a cycle is performed. In Figure 3-5, after the cycle node is determined, suppose edge d was selected. At this time, d is checked as a potential edge for extension. It can be done by checking the possibility of extension of the cycle by d. "Possibility of extension" results from the condition of p(being considered edge or cycle completing edge) $\geq$ p(any one of the cycle edges). Thus the old cycle a,b,c is extended to the new cycle a,b,d,f if the edge d is under consideration, or the cycle completing edge f satisfies the possibility of extension which is: p(d) or p(f) $\geq$ minimum of (p(a),p(b),p(c)). Now the process is continued: suppose e was selected as the next edge. But we know from the definition of the "extension of a cycle" that e cannot be considered as a potential extension because the primitive cycle d,b,e does not include the cycle node A. Hence it is discarded and the process is continued.

The next concept that we wish to explain corresponds to the relationship between a cycle and a partition. There are two cases in partitioning.

(1) Creating a partition with a new edge.

In the event that the edge selected next for inclusion (e.g. d in Figure 3-5) was not considered before, we call it a new edge. If a new edge by itself does not satisfy the possibility of extension, then we continue to check an additional new edge called cycle completing edge (e.g. f in Figure 3-5) for the possibility of extension. In Figure 3-5, new edges d and f would potentially provide such a possibility of extension of the earlier cycle formed by edges a,b,c.

If d,f meet the condition for possibility of extension stated above (namely $p(d)$ or $p(f) \geq$ minimum of $(p(a),p(b),p(c))$), then the extended new cycle would contain edges a,b,d,f. If that condition were not met, we produce a cut on edge d (called the cut edge) isolating the cycle a,b,c. This cycle can now be considered a partition.

(2) Creating a partition with a former edge.

After a cut is produced in (1), if there is a former edge, then change the previous cycle node to that node where the cut edge was incident, and check for the possibility of extension of the cycle by the former edge. For example, in Figure 3-6, suppose that a,b, and c form a cycle with A as the cycle node, and that there is a cut on d, and that the former edge w exists. Then the cycle node A is changed to C because the cut edge d originates in C. We are now evaluating the possibility of extending the cycle a,b,c into one that would contain the former edge w. Hence we consider the possibility of the cycle a,b,e,w. Assume that neither w nor e satisfies the possibility of extension, i.e., if "p(w) or p(e) $\geq$ minimum of (p(a),p(b),p(c))" is not true. Then the result is the following: i) w will be declared as a cut edge, ii) C remains as the cycle node, and iii) a,b,c becomes a partition. Alternately, if the possibility of extension is satisfied, the result is as follows: i) cycle a,b,c is extended to cycle w,a,b,e, ii) C remains as the cycle node, and iii) no partition can yet be formed.

3.4 The Graph Oriented Algorithm

An algorithm for generating the vertical fragments by the affinity graph is described below.

3.4.1 Description of The Algorithm

First we briefly describe the algorithm in 5 steps.

Step 1. Construct the affinity graph of the attributes of the object being considered. Note that the AA matrix is itself an adequate data structure to represent this graph. No additional physical storage of data would be necessary.

Step 2. Start from any node.

<u>Step 3</u>. Select an edge which satisfies the following conditions:

(1) It should be linearly connected to the tree already constructed.

(2) It should have the largest value among the possible choices of edges at each end of the tree. Note that if there are several largest values, anyone can be selected.

This iteration will end when all nodes are used for tree construction.

<u>Step 4</u>. When the next selected edge forms a primitive cycle.

(1) If a cycle node does not exist, check for the "possibility of a cycle" and if the possibility exists, mark the cycle as an affinity cycle. Consider this cycle as a candidate partition. Go to step 3.

(2) If a cycle node exists already, discard this edge and go to step 3.

<u>Step 5</u>. When the next selected edge does not form a cycle and a candidate partition exists.

(1) If no former edge exists, check for the possibility of extension of the cycle by this new edge. If there is no possibility, cut this edge and consider the cycle as a partition. Go to step 3.

(2) If a former edge exists, change the cycle node and check for the possibility of extension of the cycle by the former edge. If there is no possibility, cut

the former edge and consider the cycle as a partition. Go to step 3.

To obtain a more detailed algorithm, suppose that the following data structures are used during implementation [Bras 88]: The nodes of the affinity graph are numbered from 1 to n, $N = \{1,2, \ldots, n\}$, and a symmetric matrix L gives the weight of each edge. Three vectors are used: B, strongest, and maxwt. B gives the sequence of scanned nodes. For each node $i \in N \setminus B$, strongest[i] gives the node in B that is strongest with respect to i, and maxwt[i] gives the weight from i to strongest[i]; strongest[1] and maxwt[1] are not used. Without loss of generality we can assume that the algorithm starts from node 1. The detailed description of the algorithm now follows. The algorithm uses variables with the following meaning:

- p_cycle: a binary variable which denotes whether a primitive cycle exists
- c_node: a binary variable which denotes whether a cycle node exists
- f_edge: a binary variable which denotes whether a former edge exists
- candidate_p: a binary variable which denotes whether an affinity cycle exists which can potentially generate a fragment
- cycle_c_edge_wt: an integer variable for the weight of the cycle completing edge

- former_edge_wt: an integer variable for the weight of the former edge

```
procedure Make_partition(L[1..n,1..n]): set of edges
  { initialize flags and variables }
  B[1] <- 1
  f1 <- 1, f2 <- 0     { f1 & f2 refer to the two ends of the
                                 spanning tree }
  p_cycle,c_node,f_edge,candidate_p <- false
  pmin <-∞             { minimum edge of a cycle }
  { initialize vectors }
  for i <- 2 to n do
     strongest[i] <- 1
     maxwt[i] <- L[i,1]
     B[i] <- 0
  end_for

  repeat n-1 times
     max <- -1
     { select the next node }
     for j <- 2 to n do
        if maxwt[j] > max and
           (strongest[j] = f1 or strongest[j] = f2)
         then max <- maxwt[j]
              k <- j
     end_for

     { adjust the pointers for checking a primitive cycle }
     if strongest[k] = f1 then if f2 = 0 then f2 <- k
                                          else swap f1,f2
                                               f2 <- k
                           else f2 <- k

     { check if there is a primitive cycle }
     for j <- 2 to n do
        if B[j] = k and c_node = false
           then p_cycle <- true
                if B[j-1] > 0  then f_edge <- true
     end_for
     if p_cycle = true { primitive cycle exists }
        then { check if it is an affinity cycle }
           if pmin ≥ former_edge_wt or f_edge = false
               then candidate_p <- true
                    c_node <- true
               else f_edge <- false
        else
           insert k into B, maxwt[k] <- -∞
           if candidate_p = true
               then if maxwt[k] < pmin or
                       cycle_c_edge_wt < pmin
                    then { partition exists }
                         reinitialize variables
                         if f_edge = false
                            then save this partition
                            else change the cycle node
                                 if former_edge_wt < pmin
                                    then save this partition
                                         remove k from B
                                    else extend the cycle
                    else extend the cycle
     { pmin contains the minimum edge of a cycle }
     if pmin > maxwt[k] then pmin <- maxwt[k]

     { rearrange vectors for next selection }
     for j <- 2 to n do
        if L[k,j] > maxwt[j] then maxwt[j] <- L[k,j]
                                  strongest[j] <- k
     end_for
  end_repeat
```

3.4.2 Why does the above Algorithm Produce Reasonable Partitioning ?

Now we will give the proof of the reasonable partitioning of the algorithm. The partitioning produced by this algorithm satisfies the following properties: (1) For every cluster C_j and for every vertex $v \epsilon C_j$ and a vertex $w \notin C_j$, there exists another vertex $u \epsilon C_j$ such that $p(v,u) > p(v,w)$. Intuitively, this partitioning guarantees that for a vertex $v \epsilon C_j$, it is better to cluster v with a vertex in C_j than with a vertex outside C_j. (2) For every cluster C_j, there exists two vertices $a \epsilon C_j$, $b \notin C_j$ such that for every vertex $c \epsilon C_j$, there exists a vertex $d \epsilon C_j$ with $p(c,d) > p(a,b)$. Intuitively, this partitioning guarantees that there is a certain threshold affinity for each cluster, which is satisfied for the pairs of vertices (v_1, x_1), (v_2, x_2), ..., (v_p, x_p) where $C_j = \{v_1, \ldots,$

$v_p\}$ and $x_i \epsilon C_j$, for all $1 \le i \le p$. Note that the second property applies only when the cluster set consists of more than one cluster.

To prove these properties formally, we define the following variables.

$\pi = \{C_1, C_2, \ldots, C_r\}$, set of clusters

$p_\pi(v)$ = (i) maximum $p(u,v)$ for all $1 \le j \le r$, $v \epsilon C_j$, $u \epsilon C_j$

(ii) $p(v,v)$, if $C_j = \{v\}$

(1) States that for every (v,w) such that $v \epsilon C_j$, $w \epsilon C_k$, $j \ne k$, either $p_\pi(v) > p(v,w)$ or $p_\pi(w) > p(v,w)$.

<u>Proof</u>: Case 1: Let C_j be constructed before C_k. When vertex w is encountered for the first time in building the cycle C_k, let (v,w) be the largest affinity edge incident on w. In this case, since, when C_j was constructed, an edge (v,u), $u \epsilon C_j$, with affinity greater than (v,w) was selected as part of cycle edge, $p_\pi(v) > p(v,w)$ is true. Case 2: Let C_k be constructed before C_j. When vertex w is encountered, a cycle edge (w,x), $x \epsilon C_k$, with affinity greater than (v,w) is selected. In this case, therefore $p_\pi(w) > p(v,w)$ is true.

(2) States that for every j except k, $1 \le k \le r$, there exists a cut edge (a,b), $a \epsilon C_j$, $b \notin C_j$, such that $p_\pi(v) > p(a,b)$ for all $v \epsilon C_j$.

<u>Proof</u>: Note that there are r-1 cut edges. Since there is a unique cycle (cluster) that precedes a cut edge, it follows that each cut edge is associated uniquely with

a cluster. By the definition of extension of a cycle, each cut edge has less affinity than all the cycle edges associated with this cut edge. For example, in Figure 3-7, suppose that a,b,c and d formed a cycle and that there was a cut on edge w. Then, if p(w) had been greater than or equal to the minimum of (p(a), p(b), p(c), p(d)), then, by the definition of the possibility of extension, the cycle A,B,C,D would have been extended to the larger cycle A,B,C,D,W.

3.4.3 Examples

We will use the same example problems from Navathe et al. [Nava 84] to illustrate how this algorithm works and to compare partitioning decisions. Since our algorithm uses the same attribute affinity matrix, we assume that it has already been completed from the original transaction matrix and the computation of affinities. For ease of understanding, we will refer back to the steps of the algorithm in Chapter 3.4.1.

The attribute affinity matrix of the first example is shown in Figure 3-3 and its affinity graph after excluding the zero-valued edges appears in Figure 3-4. Suppose we start at node 9 (step 2), then, by the algorithm, edges 9-3, 9-2, and 2-8 are selected in order (step 3). At this time, edge 8-9 cannot form a cycle because it does not satisfy the possibility of a cycle (step 3). Thus edge 8-3 is selected as the next edge and it forms a candidate partition (step 4).

Note that node 3 becomes a cycle node (step 4). Then the process is continued and edge 8-7 is selected (step 3). Since there is a candidate partition, the possibility of extension is checked (step 5.1). Thus the cycle 9,3,2,8 is considered as a partition because edge 8-7 (edge being considered) and 3-7 (cycle completing edge) are both less than any of the cycle edges (step 5.1). The relevant part of the graph is shown again in Figure 3-8. As shown in Figure 3-8, our algorithm generates three affinity cycles separated by edges 3-4 and 7-8. They generate three fragments: (1,5,7), (2,3,8,9), (4,6,10). From that Figure, we know that the result of our algorithm is the same as that of Navathe et al. [Nava 84].

We conjecture that this algorithm does not depend upon the starting node. For example, let us start at node 1. By the algorithm, the first affinity cycle is not formed until edges 1-5, 5-7, 7-8, 8-2, 2-9, and 9-3 are selected. The first cycle 8,2,9,3 is identified as a candidate partition and node 8 becomes a cycle node. Then a cut occurs on edge 3-4 because neither edge 3-4 nor edge 4-8 satisfies the possibility of extension of the cycle (step 5.1). At this time, since there is a former edge, we have to change the cycle node to node 3 and check for the possibility of extension of the cycle by the former edge 7-8 (step 5.2). Thus another cut occurs on edge 7-8 because edge 7-8 and 7-3 are both less than any of the cycle edges. Figure 3-9 shows this result. Thus we can conclude that the resulting fragments are always the same irrespective of the node from which you start.

The second example we will use is a global relation with 20 attributes and 15 transactions. The result of Navathe et al. [Nava 84] partitions this relation into four fragments in three iterations: (1,4,5,6,8), (2,9,12,13,14), (3,7,10,11,17,18), (15,16,19,20). Our algorithm, however, generates five fragments in one iteration as shown in Figure 3-10: (1,5,8), (4,6), (2,9,12,13,14), (3,7,10,11,17,18), (15,16,19,20). Note that the algorithm starts from node 1 and the cut of edge 3-2 is performed earlier than that of edge 4-7. This result shows that our algorithm can find one more possibility of partitioning. Thus what the empirical objective function could not discriminate as a potential partitioning in [Nava 84], is actually detected by our procedure.

3.4.4 Complexity of the Algorithm

Now we consider the computational complexity. Step 1 does not affect the computational complexity because the attribute affinity matrix can be used as a symmetric matrix L. The repeat loop in the detailed description is executed n-1 times, where n denotes the number of attributes. At each iteration, selection of the next edge takes a time O(n). Also, whether a cycle exists or not can be implemented in time of O(n) by scanning the vector B. Thus the algorithm takes a time $O(n^2)$, which is less than that of [Nava 84], namely, $O(n^2 \log n)$.

3.5 Application and Extension

This algorithm can be used effectively for vertical partitioning because it overcomes the shortcomings of binary partitioning and it does not need any complementary procedures such as the SHIFT procedure and the CLUSTER procedure that are used in [Nava 84]. Furthermore, the algorithm involves no arbitrary empirical objective functions to evaluate candidate partitions such as those used in [Nava 84]. Also this algorithm can be used in place of the first phase without the use of any cost factors as in [Nava 84]. The second phase involving the cost-optimized vertical partitioning approach can still be applied to refine the results of the first phase.

Another important application of this algorithm is the mixed partitioning methodology that will be proposed in Chapter 5. The mixed partitioning methodology will first generate a grid for a relation vertically and horizontally, and then merge cells as much as possible by using a cost function for determining a fragment.

This algorithm can be further enhanced to address the problem of primary/secondary memory partitioning, or in the context of any memory hierarchy. By combining with the MULTI_ALLOCATE algorithm in [Nava 84], this algorithm can be used to achieve the allocation of vertical fragments over a network.

Transactions	Attributes	Predicates	Number of accesses per time period
T1	a1,a5,a7	p1,p7	25
T2	a2,a3,a8,a9	p2,p7	50
T3	a4,a6,a10	p3,p7	25
T4	a2,a7,a8	p4,p8	35
T5	a1,a2,a3,a5,a7,a8,a9	p5,p8	25
T6	a1,a5	p6,p8	25
T7	a3,a9	p5,p8	25
T8	a3,a4,a6,a9,a10	p6,p8	15

Figure 3-1 Transaction specifications

Attribute usage matrix											Type	Number of accesses per time period
Attributes / Transactions	1	2	3	4	5	6	7	8	9	10		
T1	1	0	0	0	1	0	1	0	0	0	R	Acc 1 = 25
T2	0	1	1	0	0	0	0	1	1	0	R	Acc 2 = 50
T3	0	0	0	1	0	1	0	0	0	1	R	Acc 3 = 25
T4	0	1	0	0	0	0	1	1	0	0	R	Acc 4 = 35
T5	1	1	1	0	1	0	1	1	1	0	U	Acc 5 = 25
T6	1	0	0	0	1	0	0	0	0	0	U	Acc 6 = 25
T7	0	0	1	0	0	0	0	0	1	0	U	Acc 7 = 25
T8	0	0	1	1	0	1	0	0	1	1	U	Acc 8 = 15

Figure 3-2 Attribute usage matrix

Attributes	1	2	3	4	5	6	7	8	9	10
1	75	25	25	0	75	0	50	25	25	0
2	25	110	75	0	25	0	60	110	75	0
3	25	75	115	15	25	15	25	75	115	15
4	0	0	15	40	0	40	0	0	15	40
5	75	25	25	0	75	0	50	25	25	0
6	0	0	15	40	0	40	0	0	15	40
7	50	60	25	0	50	0	85	60	25	0
8	25	110	75	0	25	0	60	110	75	0
9	25	75	115	15	25	15	25	75	115	15
10	0	0	15	40	0	40	0	0	15	40

Figure 3-3 Attribute affinity matrix

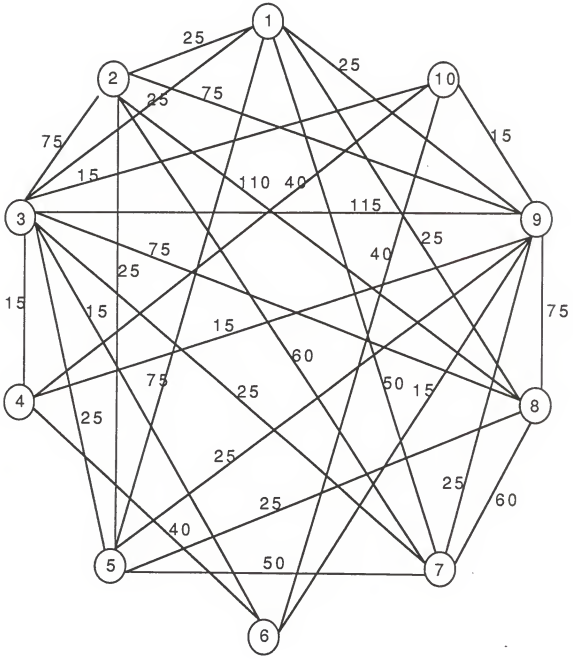


Figure 3-4 Affinity graph after excluding zero-valued edges

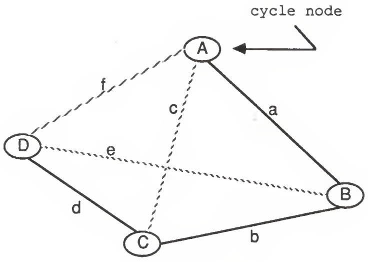


Figure 3-5   A cycle and its extension

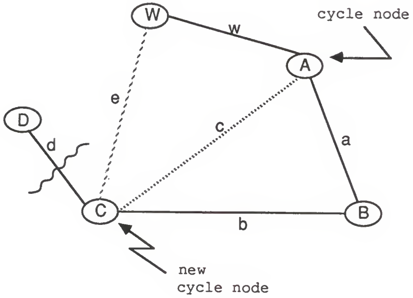


Figure 3-6   Producing a partition

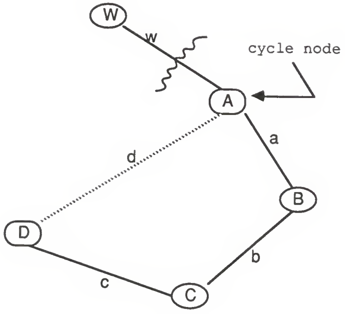


Figure 3-7 Using affinity cycle as a means for reasonable partitioning

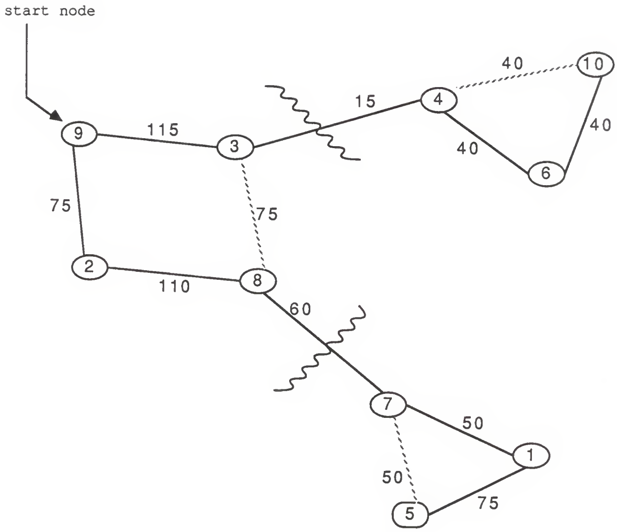


Figure 3-8 Result of the first example starting at node 9

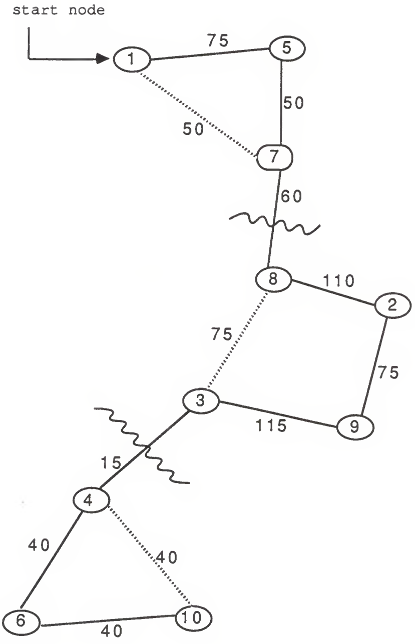


Figure 3-9 Result of the first example starting at node 1

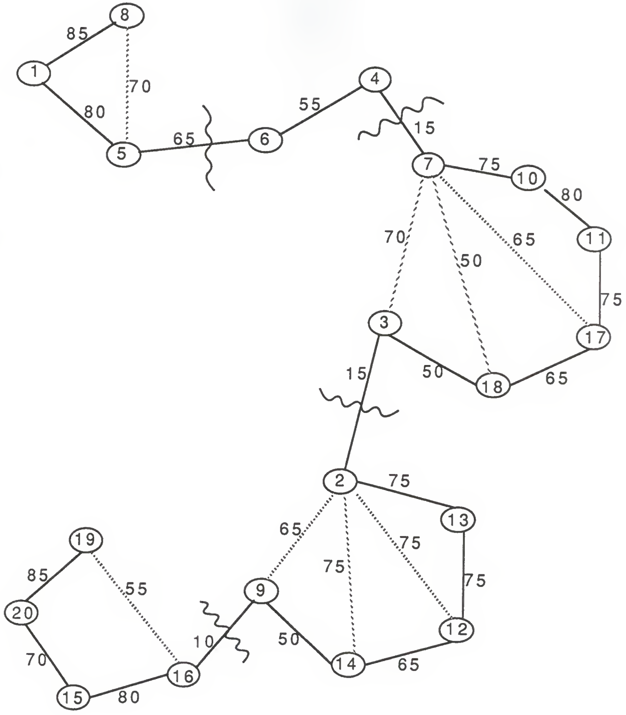


Figure 3-10 Result of the second example

CHAPTER 4
A HORIZONTAL PARTITIONING ALGORITHM

4.1 Overview

Horizontal partitioning is the process that divides a global relation into subsets of tuples, called horizontal fragments [Ceri 82, Cer 83a, Ceri 84]. Ceri, Negri and Pelagatti [Ceri 82] analyze the horizontal partitioning problem, dealing with the specification of partitioning predicates. Ceri, Navathe, and Wiederhold [Cer 83b] propose an optimization model for designing distributed database schemas with all meaningful candidate horizontal partitionings.

In the present research, we are focusing our attention on identifying all the candidate horizontal partitions. For this we propose a horizontal partitioning methodology which uses the MAKE_PARTITION algorithm presented in Chapter 3. In order to use the MAKE_PARTITION procedure of our vertical partitioning algorithm we consider only those transactions whose processing frequency is large. These transactions access tuples of the relations based on some predicates. These predicates are called simple predicates [Ceri 82]. The syntax for a simple predicate is as follows:

```
simple_predicate ::=  attribute_name <op> attribute_name
                    | attribute_name <op> value
<op> ::=  < | > | ≤ | ≥ | <> | =
attribute_name ::= string
value ::= element of the domain of the attribute.
```

A transaction gives rise to a set of simple predicates. The WHERE clause of a database operation (say SELECT) may consist of a disjunction or conjunction or negation of a set of simple predicates. Simple predicates are easier to handle and understand. Moreover, a simple predicate splits a relation into two horizontal fragments. One horizontal fragment consists of those tuples that satisfy the simple predicate and other with those tuples that do not. The correctness of fragmentation requires that each tuple of the global relation be selected in one and only one fragment.

Another consideration in horizontal partitioning is derived partitioning as stated in [Ceri 84]. Thus the predicates which give rise to derived partitioning (we call it derived predicates) should be considered in the same way as simple predicates. We limit the scope of this dissertation by assuming that all simple and derived predicates are previously determined.

As explained earlier, the focus of this research is on single relation fragmentation. Hence join predicates of the form R1.A=R2.B which deal with a pair of relations are not considered for horizontal partitioning. They are very much a part of the allocation phase and will be considered for minimizing the effort and cost of joins. In contrast, horizontal partitioning approaches used in systems like Bubba [Cope 88] or Gamma [DeW 86] attempt to achieve parallelism of join queries, etc. In our decomposition of the overall distribution problem, join predicates are in fact considered, but at a later stage.

4.2 Predicate Usage Matrix and Predicate Affinity Matrix

We use a simple example to explain our horizontal partitioning methodology below. The inputs are a set of transactions and a corresponding set of predicates as follows (assume D# and SAL are attributes of an example relation):

T1 : D#<10 (p1), SAL>40K (p7)

T2 : D#<20 (p2), SAL>40K

T3 : D#>20 (p3), SAL>40K

T4 : 30<D#<50 (p4), SAL<40K (p8)

T5 : D#<15 (p5), SAL<40K

T6 : D#>40 (p6), SAL<40K

T7 : D#<15 (p5), SAL<40K

T8 : D#>40 (p6), SAL<40K

Note that the above set of predicates do not span all the tuples of the relation; tuples with SAL=40K will not be accessed by any of the above transactions.

As the previous work [Cer 83b, Nava 84] pointed out, it is not necessary to collect information on 100% of the expected transactions (that would of course be impossible). Since the 80-20 rule applies to most practical situations, it is adequate to supply information regarding the 20% of the heavily used transactions which account for about 80% of the activity against the database.

The algorithm that we propose starts from the predicate usage matrix. The predicate usage matrix represents the use of predicates in important transactions. The predicate usage matrix for the above example (8 predicates and 8 transactions) is shown in Figure 4-1. Each row refers to one transaction: the "1" entry in a column indicates that the transaction uses the corresponding predicates. Whether the transaction retrieves or updates the relation can also be captured by another column vector with R and U entries for retrieval and update. Predicate affinity is generated in a similar manner as attribute affinity. Figure 4-2 shows a predicate affinity matrix generated from the predicate usage matrix in Figure 4-1. The numerical value of the (i,j) element in this matrix gives the combined frequency of all transactions accessing both predicates i and j and is obtained the same way as in vertical partitioning in Chapter 3. The value "==>" of the (i,j) element indicates that predicate i implies predicate j, and the value "*" represents the close usage of predicates. Two predicates i and j are "close" when the following conditions are satisfied: (1) i and j must be defined on the same attribute, (2) i and j must be used jointly with some common predicate c, (3) c must be defined on an attribute other than the attribute used in i and j. This is reasonable because predicates i, j, and c are different from one another and thus two fragments generated by predicates i,c and predicates j,c are considered "closely related" since they both involve predicate c. In the above example, p1 and p2 are "close" because of their usage with the common predicate p7 in transactions T1 and T2. These two relationships are introduced to represent logical connectivity between predicates. The attractive features of this approach are as follows:

(1) Fragments are based on actual predicates; by applying implication between predicates, the number of fragments is reduced,

(2) We can make use of the predeveloped algorithm in Chapter 3,

(3) By using clustering of predicates suggested above, a relatively small number of horizontal fragments are generated,

(4) 0-1 integer formulation is not needed.

4.3 The Algorithm

First a modified version of the graphical partitioning algorithm for clustering predicates will be given. Then by using this clustering, the entire steps for horizontal fragmentation will be described.

4.3.1 Clustering of Predicates

We will apply the graphical algorithm in Chapter 3 to horizontal partitioning. However, we cannot use this algorithm directly because the predicate affinity matrix may be too sparse. Thus we introduce two more relationships which represent logical connectivity between predicates. They are "==>" for implication and "*" for close usage as explained in Chapter 4.2.

To obtain the modified version of the graphical partitioning algorithm, the following heuristic rules are applied.

(1) A numerical value (except zero) has higher priority than the values "==>", "<==" and "*" when selecting a next edge. This is because we place more importance on affinity values which are obtained from transaction usage rather than on logical connectivity among the predicates.

(2) In the comparisons involved in checking for the possibility of a cycle or extension of a cycle, we ignore cycle edges with affinity values "==>", "<==" and "*". For example, in Figure 4-3, in comparing edge (p4,p8) with edges of the cycle (p8,p5,p6) we ignore edge (p5,p6) which has affinity "*". This is because

the affinity values "==>", "<==" and "*" represent implicit relationships and therefore can be collapsed.

(3) "==>" and "<==" are considered to have higher affinity value than "*" since the former indicates direct implication, whereas the latter represents only logical connectivity between the two predicates through their usage with a common predicate.

(4) If there are two "==>"s in a column corresponding to predicate p_k, one implied by predicate p_i and another implied by predicate p_j, then the entry (i,k) has higher priority than the entry (j,k), either if the entry (i,j) is equal to "<==", or if the entry (j,i) is equal to "==>". In other words,

$p_i \Rightarrow p_j \Rightarrow p_k$ .... (a)

$p_j \Rightarrow p_i \Rightarrow p_k$ .... (b)

in the above implication, if (a) holds, then j has higher priority than i, and if (b) holds, then i has higher priority than j.

These rules can be easily incorporated into the graphical algorithm in Chapter 3. For example, if we assume that every affinity value in predicate affinity matrix is greater than 2, then we can assign value 1 for "*" and 2 for "==>" to represent rules 1 and 3. The detailed description of the modified algorithm now follows. We use the same data structures and variables as in vertical partitioning. The meanings of variables are available from Chapter 3.

```
procedure Make_Hori_Partition(L[1..n,1..n]): set of edges
     { assign value 1 for "*" and 2 for "==>" respectively in
     a predicate affinity matrix}
  { initialize flags and variables }
  B[1] <- 1
  f1 <- 1, f2 <- 0      { f1 & f2 each refer to an end of the
                                spanning tree }
  p_cycle,c_node,f_edge,candidate_p <- false
  pmin <-∞            { minimum edge of a cycle }

  { initialize vectors }
  for i <- 2 to n do
     strongest[i] <- 1
     maxwt[i] <- L[i,1]
     B[i] <- 0
  end_for

  repeat n-1 times
     max <- -1

     { select the next node }
     for j <- 2 to n do
        if maxwt[j] > max and
           (strongest[j] = f1 or strongest[j] = f2)
         then max <- maxwt[j]
              k <- j
     end_for

     { adjust the pointers for checking a primitive cycle }
     if strongest[k] = f1 then if f2 = 0 then f2 <- k
                                          else swap f1,f2
                                               f2 <- k
                          else f2 <- k
     { check if there is a primitive cycle }
     for j <- 2 to n do
        if B[j] = k and c_node = false
           then p_cycle <- true
                if B[j-1] > 0  then f_edge <- true
     end_for

     if p_cycle = true { primitive cycle exists }
        then { check if it is an affinity cycle }
           if pmin ≥ former_edge_wt or f_edge = false
               then candidate_p <- true
                    c_node <- true
               else f_edge <- false
        else
           insert k into B, maxwt[k] <- -∞
           if candidate_p = true
               then if maxwt[k] < pmin or
                       cycle_c_edge_wt < pmin
                    then { partition exists }
                         reinitialize variables
                         if f_edge = false
                            then save this partition
                            else change the cycle node
                                 if former_edge_wt < pmin
                                    then save this partition
                                         remove k from B
                                    else extend the cycle
                    else extend the cycle

    { pmin contains the minimum edge of a cycle }
    if (maxwt[k] != 1 or maxwt[k] != 2) and  pmin > maxwt[k]
    then pmin <- maxwt[k]

    { rearrange vectors for next selection }
    for j <- 2 to n do
       if L[k,j] > maxwt[j] then maxwt[j] <- L[k,j]
                                 strongest[j] <- k
    end_for
  end_repeat
```

4.3.2 A Heuristic Procedure for Horizontal Partitioning

Now we describe the complete horizontal partitioning procedure by using the Make_Hori_Partition algorithm and the example introduced above.

Procedure HORIZ_PARTITIONING

Step 1. Perform clustering of predicates: It is done by using the Make_Hori_Partition algorithm presented in Chapter 4.3.1. We obtain in this step a set of subsets of predicates. In our example, we can obtain three clusters of predicates as shown in Figure 4-3: (p1,p7,p2), (p3,p4), (p5,p6,p8).

Step 2. Optimize predicates in each subset: In this step predicate inclusion and predicate implication are considered to minimize the number of predicates. In our example, the first subset {D#<10, D#<20, SAL>40K} is refined into {D#<20, SAL>40K} since D#<10 ==> D#<20. The second subset {D#>20, 30<D#<50} is also refined into {D#>20} since 30<D#<50 ==> D#>20, but the last one {D#<15, D#>40, SAL<40K} has no change. Note that this optimization can be done before step 1. But in this dissertation, we perform this step here in order to allow more detailed clustering. For example, a pair of predicates such as p1 and p2 in which one, (say, p1), implies another (say, p2), can be grouped in different clusters in step 1. The three clusters of predicates produced in this step, called "cluster sets", are listed in Figure 4-3.

Step 3. Compose predicate terms: Corresponding to the cluster sets (Figure 4-3) produced in step 2, we proceed as follows. The cluster sets are first evaluated to determine "the least common attribute". In our example, since SAL does not appear in cluster set 2 (corresponding to the second cluster of predicates), it is the least common attribute. Note that D# appears in all three sets. A table called the "predicate term schematic table" is now considered by placing in the first column the chosen attribute with its appropriate ranges to cover that attribute exhaustively. In our example, we create two entries: SAL<40K and SAL>40K for the SAL attribute. Then, we apply the next to least common attribute and write its appropriate ranges that appear in the cluster sets against each entry for the first column. Note that these ranges may be overlapping. In our example, D# is the next attribute. Its ranges applicable to the cluster sets are: D#<15 OR D#>40 coupled with SAL<40K (from cluster set 3), and D#<20 coupled with SAL>40K (from cluster set 1). The D#>20 predicate appearing in cluster set 2 must be written twice into the table against each entry for SAL. This resulting predicate term schematic table is shown at the top in Figure 4-4. Now we construct predicate terms from the above table as follows. Each horizontal entry in the table gives rise to one predicate term. If predicates refer to the same attributes then they are ORed, otherwise they are ANDed. The resulting predicate terms are as follows:

SAL<40K AND D#>20,

SAL<40K AND (D#<15 OR D#>40),

SAL>40K AND D#<20,

SAL>40K AND D#>20.

<u>Step 4.</u> Perform fragmentation: We have one horizontal fragment per predicate term. Thus the number of horizontal fragments will at most be the number of predicate terms plus one because there is one remaining fragment

called the "ELSE" fragment, which is the negation of the conjunction of predicate terms.

Note that the result of step 4 may be overlapped fragments or nonoverlapped fragments. Chapter 4.4 provides the ADJUST function to obtain nonoverlapped horizontal fragments.

This algorithm was implemented by using C language to demonstrate its effectiveness. It should be noted that the complexity of this algorithm is dominated by the step 1, and thus will be $O(n^2)$ for n predicates as in the vertical partitioning algorithm in Chapter 3. A smaller value of n indicates a good understanding of the heavily used predicates by users. The smaller the n, the better will be the performance of horizontal partitioning.

4.4 Nonoverlapping Horizontal Partitioning

The result of predicate partitioning may give rise to overlapped horizontal fragments. If nonoverlapped fragments are needed, then we apply the ADJUST function. For example, Figure 4-4 shows the final nonoverlapped fragments obtained by using the ADJUST function from the overlapping fragmentation generated in Chapter 4.3. There may be more than one way of creating nonoverlapped fragments from overlapped fragments. Figure 4-5 shows the three possible ways of dealing with overlapped horizontal fragments. Among these three possible ways, the best way is selected on the following criteria: (1) transaction processing cost, (2) minimization of the number of fragments. It may be desirable to choose the one that creates finer granularity for the grid cells.

A heuristic procedure for generating nonoverlapped fragments is proposed based on these criteria. Let us define the following set of variables. Here, by "part" we mean that it is a potential horizontal partition (For example, there are three parts in Figure 4-5).

i =	1,...,n denote the parts.
k =	1,...,m denote the transactions accessing the parts.
C_{ik} =	cost of accessing part i by transaction k.
F_{ik} =	the frequency with which transaction k accesses part i.
$U_{(ij)k}$=	cost of unioning two parts i and j into part ij required by transaction k.
$S_{(i/ij)k}$=	cost of selecting the i^{th} part from the merged part ij by transaction k.

The cost factors would depend on implementation details such as access methods, storage structures, use of special algorithms for duplicate elimination, etc. This level of detail is out of the present scope of our dissertation.

Using the above parameters and variables, we develop a heuristic adjusting procedure. The merging occurs if the overall cost is reduced because of some transactions accessing the two parts together.

Procedure ADJUST

<u>Step 1.</u> For each pair of contiguous parts i and j, find the cost of accessing these parts with and without merging respectively.

A. Cost without merging:

$$\sum_{k=1}^{m} C_{ik}F_{ik} + \sum_{k=1}^{m} C_{jk}F_{jk} + \sum_{k=1}^{m} U_{(ij)k}$$

B. Cost with merging:

$$\sum_{k=1}^{m} C_{(ij)k}F_{(ij)k} + \sum_{k=1}^{m} S_{(i/j)k}$$

<u>Step 2.</u> Find two parts i and j that produce the maximum saving if they are merged, and then merge them.

<u>Step 3.</u> After merging two parts, only one new part is generated. Let the set of parts after step (2) be S = 1,...,p.

<u>Step 4.</u> Then repeat steps (1)-(3) for the parts in set S till no two contiguous parts can be merged.

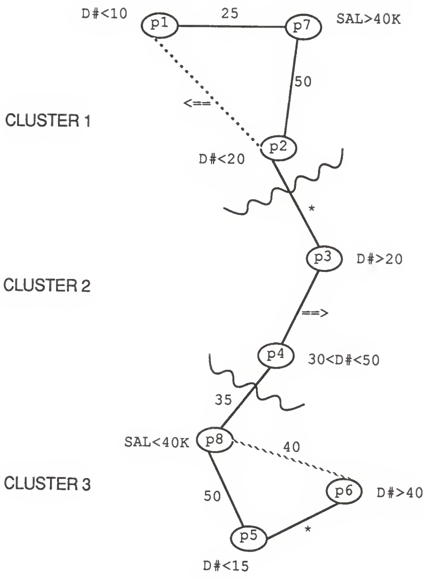
Note that if no merging occurs, then each part will be a horizontal fragment.

Predicates / Transactions	Predicate usage matrix								Type	Number of accesses per time period
	p1	p2	p3	p4	p5	p6	p7	p8		
T1	1	0	0	0	0	0	1	0	R	Acc 1 = 25
T2	0	1	0	0	0	0	1	0	R	Acc 2 = 50
T3	0	0	1	0	0	0	1	0	R	Acc 3 = 25
T4	0	0	0	1	0	0	0	1	R	Acc 4 = 35
T5	0	0	0	0	1	0	0	1	U	Acc 5 = 25
T6	0	0	0	0	0	1	0	1	U	Acc 6 = 25
T7	0	0	0	0	1	0	0	1	U	Acc 7 = 25
T8	0	0	0	0	0	1	0	1	U	Acc 8 = 15

Figure 4-1 Predicate usage matrix

Predicates	1	2	3	4	5	6	7	8
1								
2	<==,*				symmetric			
3	*	*						
4	0	0	==>					
5	0	0	0	*				
6	0	0	0	*	*			
7	25	50	25	0	0	0		
8	0	0	0	35	50	40	0	

Figure 4-2 Predicate affinity matrix



```
Cluster sets produced:   {D#<20,SAL>40K}
                         {D#>20}
                         {D#<15,D#>40,SAL<40K}
```

Figure 4-3 Clustering of predicates

PREDICATE TERM SCHEMATIC TABLE

SAL < 40K (p8)	D# > 20 (p3,p4)
	D# < 15 OR (p5) D# > 40 (p6)
SAL > 40K (p7)	D# < 20 (p1,p2)
	D# > 20 (p3,p4)
ELSE	

$\Big\downarrow$ ADJUST

NONOVERLAPPED FRAGMENTS

SAL < 40K AND D# > 20	H1 (p3,p4,p6,p8)
SAL < 40K AND D# < 15	H2 (p5,p8)
SAL > 40K AND D# < 20	H3 (p1,p2,p7)
SAL > 40K AND D# > 20	H4 (p3,p4,p7)
ELSE	H5

Figure 4-4 Nonoverlapping horizontal fragmentation

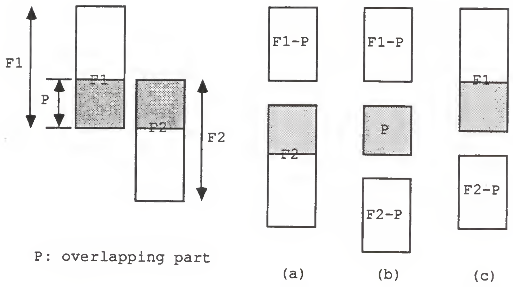


Figure 4-5 Three possible ways of dealing with overlapped horizontal fragments
(a) F2 includes P
(b) P is separated
(c) F1 includes P

CHAPTER 5
A MIXED PARTITIONING METHODOLOGY

Partitioning of a global relation in a distributed database can be performed in two different ways: vertical partitioning and horizontal partitioning. The current literature has addressed vertical and horizontal partitioning independently.

However, since database users usually access data subsets that are both vertical and horizontal partitions of global relations, there is a definite need for research on mixed partitioning. Presently there is little, if any, work on the investigation of the effects of the different sequences in which the vertical partitioning, horizontal partitioning, and mixed partitioning problems can be solved.

5.1 Proposed Methodology

As shown in Figure 1-2, the allocation problem is considered as a consequence of partitioning in our research, and thus it will be studied after partitioning problem. The mixed fragmentation approach consists of deriving a distribution design by taking into consideration the cost of processing the distributed transactions. We shall use the vertical partitioning and horizontal partitioning algorithms presented in Chapter 3 and 4 to generate a grid consisting of grid cells. These grid cells are candidate fragments as they are based on the affinity between the transactions and the data stored in the database. These grid cells may be small, and hence a transaction needs to access a set of grid cells in order to process the data. Hence, a set of grid cells in its most refined form may not give an optimal transaction processing performance. That is, the grid cells may need to be joined for the transaction to process the data. In order to minimize the number of fragments that need to be accessed by a transaction, the grid cells are considered for merging. The two major phases in the proposed mixed partitioning methodology are as follows.

GRID CREATION: This is composed of two stages; they are the vertical and the horizontal partitioning stages for grid creation. The output of the GRID CREATION is a grid corresponding to a global relation. The grid suggests all possible ways in which the global relation in a distributed database may be partitioned. Each element of the grid is called a grid cell.

GRID OPTIMIZATION: After making a grid, the GRID OPTIMIZATION performs merging of grid cells as much as possible according to the merging procedures. The merging of the grid cells is an antifragmentation procedure. Having created the cells in a top-down fashion as the ultimate smallest fragments of a relation, we now consider whether we should combine them in a bottom-up fashion. Merging is considered desirable if it reduces the overall transaction processing cost. In this dissertation two types of merging sequences are considered: vertical merging followed by horizontal merging, and horizontal merging followed by vertical merging. Thus we can calculate the cost of transaction processing at the end of each sequence respectively and then select the better of the two end results. We call the result of this phase mixed fragments.

Based on the Figure 1-2, the proposed methodology for mixed partitioning using a grid can be described in the following steps.

Procedure MIXED_PARTITIONING

Step 1. Specification of inputs: In this step, the following inputs are specified.

(1) Schema information: relations, attributes, cardinalities, attribute sizes, etc.

(2) Transaction information: name, frequency, attribute usage, etc. The attribute usage matrix is a matrix containing transactions as rows and attributes as columns. Element (i,j) = 1 if transaction i uses attribute j, else it is 0.

(3) Distribution constraints: any predetermined partitions or fixed allocation of data.

(4) System information: number of sites, transmission costs, etc. This information is used particularly to solve the allocation problem.

Step 2. Vertical partitioning for grid: In this step all candidate vertical fragments are determined. We use a graphical algorithm proposed in Chapter 3 for generating all fragments in one iteration.

Step 3. Horizontal partitioning for grid: In this step, all candidate horizontal fragments are determined. An algorithm for this step is described in Chapter 4. Note that the sequence of steps 2 and 3 can be changed.

Step 4. Merging fragments in different sequences: In this step, cells are merged as much as possible based on cost functions. The merging is carried out in two possible sequences: VH--vertical merging followed by horizontal merging, or HV--horizontal merging followed by vertical merging.

Step 5. Selection of the better sequence: Between the two different sequences, the better one is selected based on the estimation of the transaction processing cost. This selected sequence determines the final fragments of the global relation.

5.2 Grid Creation

In Chapter 3 and 4, we have seen the vertical and horizontal partitioning schemes based on the attribute and predicate affinities. We can now construct a grid consisting of cells by simultaneously applying the vertical partitioning and horizontal partitioning algorithms on the relation. Each of these cells belongs to one vertical fragment and one horizontal fragment.

Note that the cells generated by both the methods are the same. We shall use these cells for the merging procedure or grid optimization.

5.3 Grid Optimization

Based on the cost functions we want to merge cells as much as possible in order that the total cost of transaction processing (as seen from the standpoint of the one relation in question) will be minimized. Two kinds of merging are defined: horizontal merging and vertical merging. Horizontal merging deals with merging of the cells in the same row of the grid. In horizontal merging, the total number of ways of merging a fragment is $\sum_{i=1}^{n} C\ (n,i) = 2^n-1$, where C represents combination and n represents the number of fragments, because the sequence of attributes has no meaning in a relation. In our approach, however, we can minimize the possible ways of combinations of horizontal merging by using the ordered sequence of fragments generated in vertical partitioning. Thus, given n candidate horizontal fragments, a total of (n-1)+(n-2)+,...,+1 = n(n-1)/2 merging possibilities are generated. This is because, in the linearly connected spanning tree in vertical partitioning, since a cut edge between two vertical fragments is a bridge that has the maximum affinity value among all connectable edges, we can say that a fragment is more closely related to contiguous fragments compared to noncontiguous fragments. For example, in Figure 3-8, if merging is needed, fragment (1,5,7) will be merged with fragment (2,3,8,9), because fragment (1,5,7) is more related to fragment (2,3,8,9) in terms of transaction access rather than to fragment (4,6,10). This allows us to develop a heuristic procedure for horizontal merging. In vertical merging, cells in the same vertical column of the grid may be merged to produce larger fragments. Since the same graphical partitioning approach is used for horizontal partitioning, the total number of possible ways of vertical merging can be minimized in the same way as in horizontal merging.

In this dissertation, we consider both merging sequences: the VH sequence (performing vertical merging followed by horizontal merging) and the HV sequence (performing horizontal merging followed by vertical merging).

5.3.1 A Heuristic Procedure for Horizontal Merging

It should be noted that during vertical partitioning in Chapter 3 we did not take transaction processing cost into consideration, but only attribute affinity values. To that extent this is an "intuitive" affinity based partitioning.

Horizontal merging results in bringing together tuples of same id or key from two fragments. Thus for a transaction requiring two or more fragments, some saving accrues. This is a form of join where a one-to-one "matching" operation among horizontal fragments occurs based on key matching. In horizontal merging, as noted before, a fragment is merged with a contiguous fragment in the spanning tree first and then extended recursively to a larger fragment if possible.

We give below a heuristic procedure for horizontal merging. This procedure is iterative and at each iteration all the contiguous cells are considered for the possibility of merging. Those two cells which produce the maximum saving by merging are merged. This procedure stops when no two contiguous cells can be merged. We assume that the costs and frequencies considered in the formulation can be estimated. For cost estimation we use information inherent to the data and its use which is system independent, such as the length of attributes accessed and the number of tuples accessed. This approach is similar to the "bottom-up" approach used by Hammer and Niamir [Hamm 79]. They started with single attributes as the extreme vertical fragments and combined them into pairs, triplets, etc., successively. The parameters and notations for the horizontal merging are given as follows.

i =	1,...,n denote the horizontal cells.
k =	1,...,m denote the transactions accessing the horizontal cells.
C_{ik} =	cost of accessing horizontal cell i by transaction k.
F_{ik} =	the frequency with which transaction k accesses horizontal cell i.
$J_{(ij)k}$ =	cost of matching horizontal cells i and j into cell ij required by transaction k. This matching cost takes into consideration the frequency of transaction accesses to cells i and j, and the attributes accessed from the i and j "parts" of the cell ij. This cost is dependent on the transaction k.
$P_{(i/ij)k}$=	cost of projection of the ith cell from the merged cell ij by transaction k.

Then, the heuristic procedure for horizontal merging is formulated by using a greedy method as follows.

Procedure HORIZONTAL_MERGING

<u>Step 1.</u> For each pair of horizontally contiguous cells i and j, find the cost of accessing these cells with and without merging respectively.

A. Cost without merging:

$$\sum_{k=1}^{m} C_{ik}F_{ik} + \sum_{k=1}^{m} C_{jk}F_{jk} + \sum_{k=1}^{m} J_{(ij)k}$$

B. Cost with merging:

$$\sum_{k=1}^{m} C_{(ij)k}F_{(ij)k} + \sum_{k=1}^{m} P_{(i/ij)k}$$

Step 2. Find two cells i and j that produce the maximum saving if they are merged, and then merge them.

Step 3. After merging two cells horizontally, only one new cell is generated. Let the set of resultant merged cells after step (2) be I = 1,...,p.

Step 4. Then repeat steps (1)-(3) for the cells in set I till no two contiguous cells can be merged.

Note that this merging procedure is obviously dominated by step 4. Other steps take a time of O(n), where n represents the number of fragments. Thus the complexity of this merging procedure is $O(n^2)$.

5.3.2 A Heuristic Procedure for Vertical Merging

Vertical merging is very similar to horizontal merging. Note that we performed graphical top-down decomposition based on the notion of affinities. But affinity is a heuristic. Cost-based optimization gives further improvement. We achieve this by merging. The merging occurs if the overall cost is reduced because of some transactions accessing the two cells together. Let us define the following set of variables:

$i =$ 1,...,n denote the vertical cells.

$k =$ 1,...,m denote the transactions accessing the vertical cells.

$C_{ik} =$ cost of accessing vertical cell i by transaction k.

$F_{ik} =$ the frequency with which transaction k accesses vertical cell i.

$U_{(ij)k} =$ cost of union of two vertical cells i and j into cell ij required by transaction k.

$S_{(i/ij)k} =$ cost of selection of the ith cell from the merged cell ij by transaction k.

Using the above parameters and variables, we develop a heuristic procedure for vertical merging in a similar manner as in the horizontal merging.

Procedure VERTICAL_MERGING

<u>Step 1.</u> For each pair of vertically contiguous cells i and j, find the cost of accessing these cells with and without merging respectively.

A. Cost without merging:

$$\sum_{k=1}^{m} C_{ik}F_{ik} + \sum_{k=1}^{m} C_{jk}F_{jk} + \sum_{k=1}^{m} U_{(ij)k}$$

B. Cost with merging:

$$\sum_{k=1}^{m} C_{(ij)k}F_{(ij)k} + \sum_{k=1}^{m} S_{(i/ij)k}$$

Step 2. Find two cells i and j that produce the maximum saving if they are merged, and then merge them.

Step 3. After merging two cells vertically, only one new cell is generated. Let the set of resultant merged cells after step (2) be I = 1,...,p.

Step 4. Then repeat steps (1)-(3) for the cells in set I till no two contiguous cells can be merged.

Note that this merging procedure is obviously dominated by step 4 as in horizontal merging. Other steps take a time of O(n), where n represents the number of fragments. Thus the complexity of this merging procedure is also $O(n^2)$.

5.3.3 An Example

In our example we use an access factor α that reflects the cost of processing a merged fragment relative to the total cost of accessing its constituent cells if they are not merged together. Thus α is defined as the ratio of the cost of accessing a merged fragment to the cost of accessing its constituent grid cells. Detailed estimation of α would depend upon implementation factors, particularly the access structures such as indices and the access methods used.

In Figure 5-1 we show the mapping of the transactions to the grid cells. Note that some of the grid cells are not accessed by the set of most important transactions, but may be accessed by other transactions. The mapping is done by mapping attributes and the predicates of the transactions with the attributes and the predicates forming the grid cells. For example, transaction T4 accesses attributes a2, a7 and a8, and is based on predicates p4 and p8, whereas the grid cell G1 is formed of attributes a1, a5 and a7, and predicates p3, p4, p6 and p8. Hence transaction T4 accesses the cell G1.

Figure 5-2 shows the costs of accessing each of the horizontal fragments H1, H2, H3, H4 and H5. They are 100, 150, 200, 75 and 125 respectively. The length of the attributes of each of the vertical fragments are 14, 20 and 16 respectively. Note that we assume a linear cost access model in which the cost of accessing a grid cell is proportional to the length of attributes in that cell. Hence the cost of accessing a single grid cell G5 is C5 = 150 * 20/(14+20+16) = 60.

Now we give an example that shows how two contiguous cells are merged. In the following illustration, we use the join cost factor as a variable to show how merging is dependent upon the join cost. Let us consider the two contiguous cells G1 and G2 in Figure 5-2. The parameter α is assumed to be 1.2 and the projection cost is assumed to be 0.1*J.

Cost without merging:

$$\sum_{k=1}^{8} F_{1k}C_{1k} = F_{14}C_{14} + F_{16}C_{16} = 35 * 28 + 25 * 28 = 1680$$

$$\sum_{k=1}^{8} F_{2k}C_{2k} = F_{24}C_{24} + F_{28}C_{28} = 35 * 40 + 15 * 40 = 2000$$

$$\sum_{k=1}^{8} F_{(12)k}J_{12} = F_{(12)4}J_{12} + F_{(12)6}J_{12} + F_{(12)8}J_{12}$$
$$= 35*J + 0 + 0$$
$$= 35*J$$

Note that J_{12} is the join cost for the grid cells G1 and G2, and the only transaction T4 accesses the merged fragment with frequency 35.

Cost with merging:

$$\sum_{k=1}^{8} F_{(12)k}C_{(12)k} = F_{(12)4}C_{(12)4} + F_{(12)6}C_{(12)6} + F_{(12)8}C_{(12)8}$$
$$= 35 * 81.6 + 25 * 81.6 + 15 * 81.6$$
$$= 6120$$

$$\sum_{k=1}^{8} F_{1k}P_{1/12} + \sum_{k=1}^{8} F_{2k}P_{2/12} = F_{16}P_{1/12} + F_{28}P_{2/12}$$
$$= 25 * 0.1*J + 15 * 0.1*J$$
$$= 4*J$$

where $C_{12} = \alpha * (C_1 + C_2) = 1.2 * (28 + 40) = 81.6$

Thus we can conclude that the two cells are merged if 1680 + 2000 + 35*J > 6120 + 4*J, that is if 31*J > 2440 or J > 78.7. Figure 5-3 shows one of the possible results of grid optimization. The cost factor J generally would depend upon a variety of factors including join selectivity, implementation details, and join algorithms.

5.3.4 Result Comparison by Total Cost Computation

The above two procedures for horizontal and vertical merging are independent and can be applied in any sequence. The end result of both can be subjected to a cost computation against the given set of transactions to pick the better of the two sequences (i.e. HV and VH).

The total cost for the process of merging cells is thus calculated in the following way.

$$C_T = \sum_{i=1}^{n} \sum_{k=1}^{m} COST_{ik}$$

where n : number of final fragments

m : number of transactions accessing the final fragments

C_T : total cost

$COST_{ik}$: cost for accessing a resulting fragment i (which is given at the end of the merging procedure) by transaction k.

Note that at the end of this step we have achieved a fragment configuration which could not be obtained by either an HV or VH sequence of partitioning independently. The merging steps are necessary to achieve the meaningful combinations of grid cells to minimize the total processing cost. The resulting set of cells is then evaluated together with allocation (with and without redundancy) later.

5.4 Effect of Changes in Data and Transactions

In the case of new transactions we have to do all the five steps given in the MIXED_PARTITIONING procedure in Chapter 5.1. In the case of the same schema and the same transactions, but new data tuples, only the cost parameters will be affected for the grid optimization part. Hence only step 4 of the MIXED_PARTITIONING procedure has to be redone for merging the fragments. This is an advantage of this methodology. In the case of adding new relations to the schema, if existing transactions are not modified to access these new relations, an analysis based on the old set of transactions will not reveal any new result. However, if we incorporate new transactions that use the new relations, we have to consider all the five steps of the MIXED_PARTITIONING procedure.

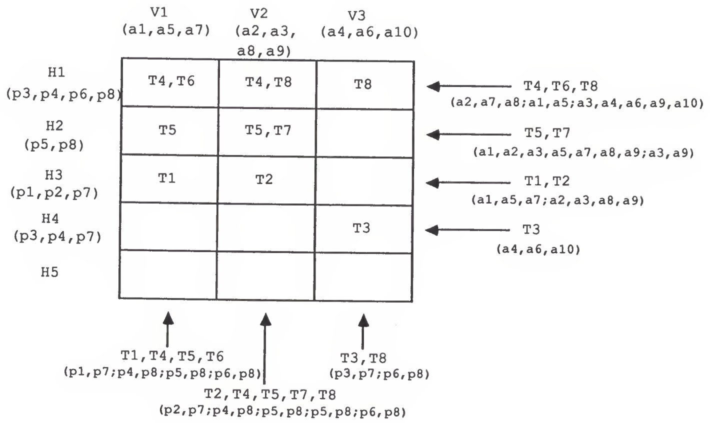


Figure 5-1 Mapping from transactions to grid cells

Cost	Attributes length		
	14	20	16
100	G1	G2	G3
150	G4	G5	G6
200	G7	G8	G9
75	G10	G11	G12
125	G13	G14	G15

$$C_i = \text{cost of accessing the corresponding horizontal fragment} * \frac{\text{L(attributes of the corresponding vertical fragment)}}{\text{L(all attributes)}}$$

e.g. $C5 = 150 * \frac{20}{14+20+16} = 60$

Figure 5-2 Cost model for merging grid cells

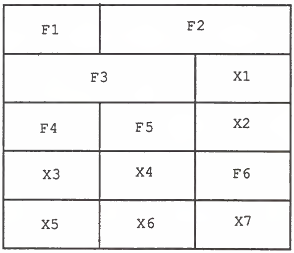


Figure 5-3 A possible result of grid optimization

CHAPTER 6
AN ALLOCATION ALGORITHM FOR THE MIXED FRAGMENTS

In a distributed database system, a user accesses the system through a local processor. The system may make local and remote accesses functionally transparent. So data allocation is one of the most important distributed database issues [Cer 83b, Corn 88]. The problem of data allocation in distributed systems has been extensively studied together with file assignment problem. Even though this problem is closely related to the file assignment problem, it differs greatly from the file assignment problem because, in the distributed database systems, the way the data are accessed is far more complex [Aper 88].

Data allocation is the process of mapping each logical fragment to one or more sites. Previous research in this area has been performed in two ways: Data allocation by itself, and extension of the pure data allocation problem by including the network topology and communication channels in the decision variables. This dissertation is concerned with pure data allocation in which the unit of allocation is the mixed fragment which comes from our mixed partitioning procedure in Chapter 5 that considers both horizontal partitioning and vertical partitioning simultaneously. The objective of our approach is to allocate the mixed fragments nonredundantly with minimum distributed transaction processing cost. To achieve this we present some heuristics that combine fragment allocation and transaction processing strategy. Based on the heuristics we develop an algorithm not by using the 0-1 integer programming method, but by using a graphical method called allocation-request graph. Note that in mixed fragment allocation there may be a lot of different types of fragments. Thus we define three more operations in addition to the join operation for these mixed fragments.

This approach begins from the transaction analysis which produces operation trees and a fragment usage matrix. Then, by using this fragment usage matrix, an allocation-request graph is generated, and finally fragments are allocated according to the heuristic allocation procedure.

6.1 Distributed Transaction Processing

In data allocation problems, data should be locally processed by applying the conditions of the query before any transmission. The selections and projections, which do not require data movement should be performed locally. The really difficult problem of distributed query optimization is the execution of joins.

When the unit of allocation is the mixed fragment, since there are no restrictions on the type of fragments, in addition to the join operation, there are other operations which require data movement: outer-join, union, and outer-union. We call them multi-fragment operations. Outer-join is introduced for keeping all tuples in fragments regardless of whether or not they have matching tuples in the other fragment, and outer-union is introduced for keeping nonunion compatible attributes.

In a distributed system there are several factors that must be taken into account such as link cost, data transmission cost, storage cost, IO cost, and CPU cost. Among them, the first and the most important factor to be considered in distributed environments is the cost of transferring data over the network [Wied 87]. Thus, in this dissertation we minimize the amount of data transmission, which we call the total transmission cost, while satisfying the CPU cost constraint at each site. The details will be explained in Chapter 6.2. Note that we assume that the link transmission costs between any two nodes are the same.

When a multi-fragment operation is related to the fragments that reside at different sites, the query processing strategy on site selection for performing this operation can have a significant impact on data movement. There are basically two alternatives for query processing strategy [Elma 89]. One is the site-of-origin strategy in which all multi-fragment operations are performed at the site where the query was submitted; the other is the move-small strategy in which for each multi-fragment operation, the smaller fragment participating in the operation is always sent to the site where the relation of larger size resides in order to perform multi-fragment operation. The move-small strategy was shown to be superior to the site-of-origin strategy [Corn 88]. We will illustrate this with a simple example query. Suppose fragments F1 and F2 are as shown in Figure 6-1. Consider the query Q: "For each employee retrieve the employee name and salary". Then each transaction against a relational database can be decomposed into a sequence of relational algebra operations.

Suppose the query was submitted at a distinct site 3. Neither the F1 nor the F2 fragments reside at site 3. There are two simple strategies for executing the distributed query:

(1) Data transfer in site-of-origin strategy
Transfer both the F1 and F2 fragments to the site 3 and perform the query at site 3. In this case we need to transfer a total of 1000*50 + 100*30 = 53000 bytes.

(2) Data transfer in move-small strategy
Transfer the F2 to site 1, execute the query at site 1, and send the result to site 3. The size of query result is 36*1000 = 36000 bytes, so we transfer 36000 + 3000 = 39000 bytes.

However, in the move-small strategy the multi-fragment operation is performed at the site where the larger fragment is located. The result of the multi-fragment operation should be returned to the site which requests the multi-fragment operation if the larger fragment participating in the multi-fragment operation is not in that site. Thus we have to determine the query processing strategy in order to minimize the amount of data transmission. However, even though we consider the transaction processing strategy to reduce the total transmission cost, it is difficult to obtain optimal allocation by only considering transaction processing strategy because optimal allocation is deeply related to both transaction processing strategy and fragment allocation. Actually, finding a nonredundant, minimum total data transmission cost allocation is NP-complete [Gare 79]. Hence, to minimize the transmission cost, we will consider some heuristics which allow multi-fragment operations to be performed effectively in an integrated way.

In our research, to illustrate how a transaction is decomposed into a sequence of algebraic operations, we use an operation tree which represents algebraic formulation. Figure 6-2 shows an example of an operation tree. Note that at this step we assume that the amount of data requested by each transaction is estimated. It can be done by considering attributes accessed and selectivity factors. After analyzing transactions, a fragment usage matrix is generated. Fragment usage matrix is a matrix which is similar to the attribute usage matrix, and represents the use of the fragments in important transactions. Each row refers to one transaction:

the numeric value in a column indicates that the transaction originated from the specified site or sites (numeric value) and "uses" the corresponding fragment. For example, in Figure 6-3, transaction T5 originates from the sites 1 and 2 and uses fragment F3.

6.2 Representation and Strategy

6.2.1 Representation

To allocate fragments optimally most approaches [Cer 83b, Corn 88] use the 0-1 integer programming method. However, this method is hard to understand and lacks intuition. It also suffers from limitations on the size of the problem to solve with commercially available software. In this dissertation we use a graph called allocation-request graph to represent how data are requested and allocated. Figure 6-4 shows an example of the allocation-request graph. As shown in Figure 6-4, an allocation-request graph consists of three components.

(1) rectangular nodes, for fragments

(2) circular nodes, for sites

(3) edges, that denote requests by the transactions for accessing the fragments

Edges, which are directed, are labeled with a quadruple (i,j,f,d), where d stands for the amount of data requested from the fragment j that corresponds to the rectangular nodes in the graph for processing the i^{th} transaction, and f stands for the frequency with which this transaction is executed. For example, in Figure 6-4, the label (2,1,30,200) means that transaction 2 requests 200 bytes of fragment 1, with frequency of 30.

Join operations are represented by drawing arcs among edges participating in joins and marking them with a "J". Outer-union operations and outer-join operations are represented by "O", and union operations are represented by "U". Figure 6-4 shows these representations. Note that n-way (n>2) multi-fragment operations can also be represented in the same way as shown in Figure 6-4.

6.2.2 Strategy

The objective is to allocate the mixed fragments nonredundantly with minimum distributed transaction processing cost. The distributed transaction processing cost here is measured by the total transmission cost. The allocation and transaction processing are subject to some load balancing constraints. The total transmission cost is obtained by multiplying transaction frequency by the amount of data requested. To reduce the total transmission cost, a fragment should be allocated to the site that requests the largest amount of data. To increase the availability of the system for local processing, CPU/IO cost is considered as a balancing constraint at each site. This means that there is an upper limit of the CPU/IO time at each site. The CPU/IO cost is determined by (cost of tuple retrieval and processing) x (# of tuples) x (# of retrievals). This is because in most relational database systems they use a tuple at a time retrieval storage system (e.g. RSS in System R). Thus what we are trying to do is to minimize the total transmission cost while satisfying the CPU/IO cost constraint at each site.

This objective would be easily obtained if transactions did not contain multi-fragment operations. However, if transactions contain multi-fragment operations, as it would normally happen, we cannot easily figure out the objective because there is an interdependency between query processing strategy and data allocation as pointed out in the previous approaches [Wah 84, Aper 88, Corn 88]. To solve this problem data allocation and query processing strategy should be considered together. This means that we should choose the sites where data are to be stored while simultaneously determining where join operations should take place [Corn 89].

Another consideration is the priority of the requests. When there are multiple requests for a fragment, we emphasize on multi-fragment requests for selecting a candidate request. This is because it allows effective processing for multi-fragment operations. Note that nonmulti-fragment operations are not neglected. It means that the relationship with the other fragment participating in a multi-fragment operation is considered to reduce the data transmission. The details will be explained in strategy 3. This is reasonable because, as pointed out in [Swam 89], some future applications built on top of relational systems will require processing of queries with a much larger number of joins. Object-oriented database systems that use relational systems for information storage are another class of potential applications performing many joins.

Thus, in our research, we present an integrated heuristic that takes into consideration the interdependency problem in multi-fragment operations and join emphasis for selecting a candidate request. The integrated heuristic is as follows: "Let the fragment that causes the largest data transmission be located at the requesting site as much as possible while satisfying the load balancing constraints." This heuristic can be accomplished by using an allocation technique called pseudoallocation, which allows fragment allocation dynamically with multi-fragment operations to minimize the distributed transaction processing cost. This will be explained later.

Based on the above discussions we can formulate the following allocation strategies. Note that at any step the final allocation of a fragment is determined as long as the load balancing constraint is not violated, otherwise the fragment goes back to the fragment pool.

<u>Strategy 1.</u> If there is only a single request, allocate this fragment to the requesting site.

Strategy 2. When there are multiple requests from different sites to one fragment, which are not participating in any multi-fragment operations, allocate this fragment to the site that gives rise to the largest amount of data transmission.

Strategy 3. If the requests are mixed with multi-fragment operations, two cases may occur. To explain further, we use the following terminology.

- former fragment: the fragment that is being considered for allocation first between the two fragments participating in a multi-fragment operation. For example, in Figure 6-5, F1 is a former fragment.
- latter fragment: the fragment that is being considered for allocation after the former fragment. In Figure 6-5, F2 is a latter fragment.
- data_multi_former (data_multi_latter): the total amount of data requested of the former fragment (latter fragment) from the site that has multi-fragment operations. Note that if there are several sites that have multi-fragment operations then the site that has the largest amount of total data requested is considered. In Figure 6-5, a and c represent data_multi_former and data_multi_latter respectively.
- data_nomulti_former (data_nomulti_latter): the total amount of data requested of the former fragment (latter

fragment) from the site that has no multi-fragment operations. Note that if there are several sites that have no multi-fragment operations then the site that has the largest amount of total data requested is considered. In Figure 6-5, b and d represent data_nomulti_former and data_nomulti_latter respectively.

3.1 Former fragment: If the candidate fragment is a former fragment, then the requests that are participating in multi-fragment operations have higher priority for allocation without cost computation than those that are not participating in multi-fragment operations. Thus, in Figure 6-5, fragment F1 will be allocated to site 1 without any cost computation.

3.2 Latter fragment: If the candidate fragment is a latter fragment, then apply the pseudoallocation technique.

6.2.3 Pseudoallocation Technique

Pseudoallocation technique is an allocation heuristic that allows fragments to be allocated dynamically with multi-fragment operations. The job of the pseudoallocation technique is to determine whether a former fragment is kept in the allocated site or not. Thus the pseudoallocation technique is always meaningful when there is a former fragment. In order to satisfy our proposed heuristic we introduce a parameter β to represent the ratio of the result size of multi-fragment operations to the size of the larger of the two participating fragments. Thus if both the fragments participating in a multi-fragment operation are allocated to remote sites, some portion of a fragment depending on the ratio will be added to the total data transmission as a penalty for the remote accessing. The rules are as follows. We refer to Figure 6-5.

(1) $a<b$ & $c<d$

- Meaning: site S1 requests less data from F1 and F2 than site S2.
- Additional condition
  - (a) $a>c$ & $\beta*a>b$
    - Meaning: data requested from F1 is greater than data requested from F2 in the multi-fragment operation and data resulting from multi-fragment operation due to F1 is greater than data requested by S2 from F1.
    - Decision: keep F1 at S1 and return F2 to the fragment pool. Note that, however, if $\beta*a<b$ then the two fragments are returned to the fragment pool.
  - (b) $a<c$ & $\beta*c>d$
    - Meaning: data requested from F1 is less than data requested from F2 in the multi-fragment operation and data resulting from multi-fragment operation due to F2 is greater than data requested by S2 from F2.
    - Decision: release F1 and allocate F2 to S1. Note that if $\beta*c<d$ then the two fragments are returned to the fragment pool.

(2) $a<b$ & $c>d$

- Meaning: S1 requests less data from F1 than F2 and S2 requests less data from F2 than S1.
- Decision: release F1 and allocate F2 to S1.

(3) $a>b$ & $c<d$

- Meaning: S1 requests more data from F1 than F2 and S2 requests more data from F2 than S1.
- Decision: keep F1 at S1 and return F2 to the fragment pool.

(4) $a>b$ & $c>d$

- Meaning: S1 requests more data from F1 and F2 than S2.
- Additional condition

  (a) $a>c$

  - Meaning: data requested from F1 is greater than data requested from F2 in the multi-fragment operation.
  - Decision: keep F1 at S1 and return F2 to the fragment pool.

  (b) $a<c$

  - Meaning: data requested from F1 is less than data requested from F2 in the multi-fragment operation.
  - Decision: release F1 and allocate F2 to S1.

Now let us explain how the pseudoallocation technique works. For example, in Figure 6-6, suppose F2 is the former fragment and β=1.5. Then, according to the strategy 3.1, fragment F2 is allocated to site 1 with data_multi_former <

data_nomulti_former. Then we would like to allocate the next fragment F3 with data_multi_latter < data_nomulti_latter. Since data_multi_former < data_multi_latter & β*data_multi_latter > data_nomulti_latter, the latter fragment F3 is allocated to site 1 and the former fragment F2 is kicked off from site 1 by strategy 3.2. Fragment F2 is reconsidered and allocated to site 2 if it satisfies the load balancing constraints. However, suppose that data_multi_latter < 4500. Then the former fragment F2 will still be at site 1 and the latter fragment F3 will go back to the fragment pool for reconsideration. This means that allocation of a fragment participating in a multi-fragment operation is confirmed after considering the effect of all the multi-fragment operations. In the next chapter we will give a heuristic allocation procedure.

6.3 A Heuristic Procedure for Mixed Fragment Allocation

Now a heuristic procedure for allocating the mixed fragments by the allocation-request graph is described below. First we briefly describe the algorithm in three steps.

Step 1. Form an allocation-request graph by using operation trees and a fragment usage matrix.

Step 2. Sort fragments based on some reasonable sorting criterion (e.g. size of fragment, frequency of retrieval or the ratio of frequency over size). Place the result in a circular queue.

Step 3. While the queue is not empty do

(1) take the front fragment,

(2) allocate it to an appropriate site according to the allocation strategies,

(3) apply the pseudoallocation technique as defined and explained in Chapter 6.2.3,

This iteration will end when all fragments are allocated.

To obtain a more detailed algorithm, suppose that the following data structures are used during implementation: The fragments are numbered 1 to n. An adjacency list L represents the allocation-request graph, in which the array HEAD[1..n] contains fragment_id, total amount of data requested to the fragment, number of requests to the fragment, and number of multi-fragment operations in the list pointed by this fragment. Each list has five pieces of information: transaction_id, frequency, data requested, site_of_origin and type of transaction. The detailed description of the algorithm now follows. The algorithm uses more variables with the following meaning in addition to those in Chapter 6.2.

- #_of_req: an integer variable for the number of data requests to the fragment.
- #_of_mulop: an integer variable for the number of multi-fragment operations in the list pointed by the fragment.

- pre_allo(i): an integer vector which denotes the pre-allocation site without cost computation for fragment i.
- confirm(i): a boolean vector which denotes the status of final allocation of fragment i.

```
Procedure Mixed_F_Allo (adjacency list L with HEAD[1..n])
   {prepare a circular queue}
   sort HEAD array based on the sorting criteria and make the
   result as a circular queue.


   while queue is not empty do
      take the front one from the queue (we call it fragment
      i)
      if #_of_req = 1 {there is a single request}
      then  check the balancing constraints and allocate this
            fragment to the requesting site.
            confirm(i)=true
      else   {there are multiple requests}
      case #_of_mulop = 0: {no requests are participating in
                                 multi-fragment operations}
               check the balancing constraints and allocate
               this fragment to the site that gives rise to
               the largest data transmission.
               confirm(i)=true

           #_of_mulop >= 1 & pre_allo(k)=empty for all k:
                 { it is a former fragment}
               If    if all multi-fragment operations are from
                     one site
               then  check the balancing constraints and
                     allocate the fragment to the requesting
                     site.
                     pre_allo(i) <- site_id
               else  determine the allocation site based on
                     the cost computation and the balancing
                     constraints.
                     pre_allo(i) <- site_id

           #_of_mulop >= 1 & pre_allo(k) is not empty:
               {it is a latter fragment}
               find the former fragment and call it j.
               check the balancing constraints and allocate
               the fragment to the site where the former
               fragment resides.
               if data_multi_former ≤ data_nomulti_former
```

```
                then if data_multi_latter ≤ data_nomulti_latter
                   then if data_multi_former ≥data_multi_latter
                         then  determine the former fragment's
                               site using β.
                               insert the latter into queue
                         else  determine the latter fragment's
                               site using β.
                               insert the former into queue
                   else  insert the former into queue
                         confirm(i)=true
                else if data_multi_latter ≤ data_nomulti_latter
                   then  insert the latter into queue
                         confirm(j)=true
                   else if data_multi_former ≥
                                  data_multi_latter
                         then  confirm(j)=true
                               check the balancing constraint
                               for the latter fragment.
                         else  confirm(i)=true
                               check the balancing constraint
                               for the former fragment.
      end_case
   end_while
end_procedure
```

6.4 Complexity of the Algorithm

Now let us analyze the time complexity of the above algorithm based on the steps. Obviously step 2 takes a time of O(nlogn), where n means the number of fragments, because it needs sorting. Step 3(b) requires the request-list for the fragment to be traversed, so that it takes a time of O(m), where m represents the number of transactions. Step 3(c) takes a time of O(m) because finding a former fragment needs a traversal for a fragment and its list. The "while" loop will run m/n times per fragment, giving a time in O(m). This is because, by using a flag, a fragment is allocated only once per site before final allocation. This flag guarantees the termination of the algorithm. Thus step 3 takes a time of $O(m^2)$, and the time required by this algorithm is therefore in $O(m^2)$.

6.5 Experimental Results

In order to show how the proposed heuristic procedure indeed works as claimed and how the pseudo allocation technique can be used for obtaining optimal solutions, we implemented this algorithm by using the C language. To simplify the implementation we assume the following: (1) All multi-fragment operations are 2-way operations, (2) If both fragments are allocated to remote sites then these sites are distinct, (3) Load balancing consideration is deferred.

We use three examples, namely EX1, EX2, and EX3 as shown in Figure 6-6, 6-7, and 6-8 respectively. EX1 has 3 sites and 3 fragments, EX2 3 sites and 6 fragments, and EX3 4 sites and 6 fragments. Figure 6-9 shows some experimental results. We varied the parameter β from 0.5 to 2.0. As explained before, β represents the ratio of the result size of multi-fragment operations. We computed the amount of data transmission of the optimal allocation scheme by exhaustive search, and compared it with the heuristic allocation scheme. The optimality of the algorithm is dependent on the parameter β. It appears that as β increases, the heuristic solution tends to equal to the optimal solution. This approach assumed an average β. This can be extended to pairwise β values.

It should be noted that we have been considering the accessed fragments so far. Now we are going to discuss the allocation of the nonaccessed fragments.

6.6 Dealing with Nonaccessed Fragments

Let us consider the fragments that are not accessed by important transactions. As shown in Figure 5-3, some of fragments are accessed by the important transactions of 20%, whereas some of fragments are not accessed. However, this does not mean that these nonaccessed fragments are not accessed at all because the 80% of less important transactions may still be accessing those fragments. Thus when we allocate fragments, we have to consider the allocation of these nonaccessed fragments. The problem is caused by the fact that we do not know the nature of the 80% of less important transactions. Based on the current inputs, the following solutions can be possible:

(1) Merging with a contiguous accessed fragment

When we consider merging a nonaccessed fragment with a accessed fragment, the merging will be in the direction that the result will remove the outer-join and outer-union to reduce the multi-fragment operation cost. This is because outer-join and outer-union operations need more cost because of padding information. For example, in Figure 5-3, consider fragments F5 and F6, and cells X2 and X4. Here at least there are two choices in merging. One way is that cell X2 and X4 are merged with F5, and the other way is that cell X2 is merged with F5 and cell X4 is merged with F6. Then it is obvious that the latter way is better than the former way in the context of transaction processing cost because the former way needs outer-union.

(2) Allocation to the site that has the cheapest storage cost

Suppose there was a site whose storage cost is the cheapest one, then we can allocate all nonaccessed fragment to that site.

(3) Random allocation

Basically the nonaccessed fragments are considered as don't care conditions. It does not matter where this nonaccessed fragments are allocated.

EMPLOYEE relation

Site 1 : F1

ENO	FNAME	LNAME	SEX	BDATE	MGRSSN

1000 records
Each record has 50 bytes long
ENO field is 4 bytes long
FNAME field is 15 bytes long
LNAME field is 15 bytes long

Site 2 : F2

ENO	ADDR	SAL

100 records
Each record has 30 bytes long
ENO field is 4 bytes long
SAL field is 6 bytes long

Figure 6-1 Example to illustrate data transfer

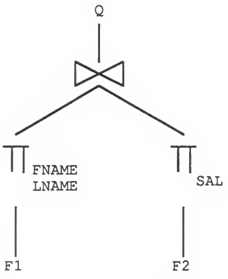


Figure 6-2 Operation tree

transactions \ fragments	F1	F2	F3	F4	F5	F6	Number of accesses per time period
1				1,2			Acc 1 = 25
2					2		Acc 2 = 50
3						3	Acc 3 = 25
4	1	1	1				Acc 4 = 35
5			1,2				Acc 5 = 25
6	2						Acc 6 = 25
7			3				Acc 7 = 25
8		3					Acc 8 = 15

Figure 6-3 Fragment usage matrix

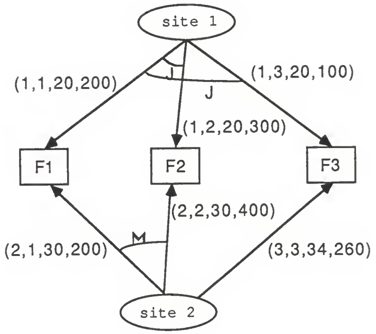


Figure 6-4 Allocation-request graph

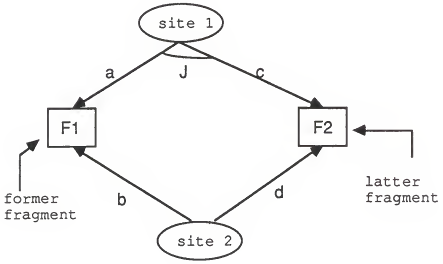


Figure 6-5 Example to explain terminology

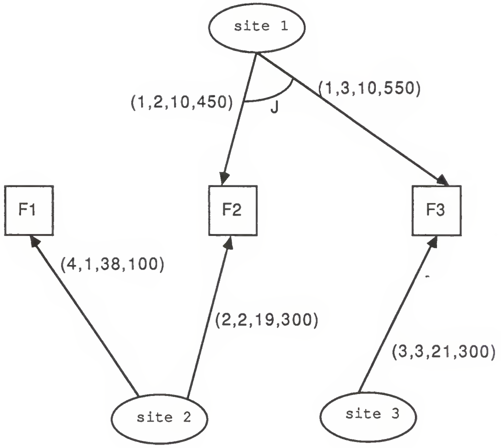


Figure 6-6   Example 1

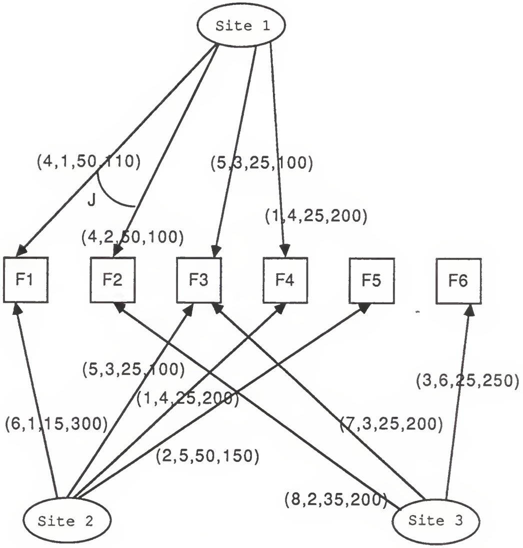


Figure 6-7 Example 2

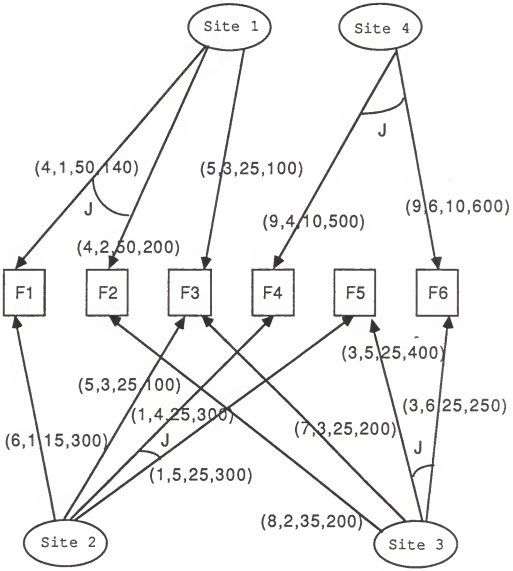


Figure 6-8 Example 3

example / ß	ex1		ex2		ex3	
	heuristic	optimal	heuristic	optimal	heuristic	optimal
0.5	7250	7250	19500	17750	35250	32000
1.0	10000	10000	19500	19500	35250	35000
1.5	10800	10800	19500	19500	35250	35250
2.0	10800	10800	19500	19500	35250	35250

Figure 6-9 Experimental results

CHAPTER 7
ALTERNATIVE WAYS OF DEALING WITH FRAGMENTATION AND ALLOCATION

In this dissertation, we presented a set of distributed database design methodologies. In our approach, mixed fragmentation involves a combined use of horizontal and vertical partitioning to construct grid cells. These cells are then merged during grid optimization. Allocation is deferred as a separate problem that is governed by the site of origin of transactions. We describe the other approaches in the next sub chapter.

7.1 Alternatives for Combining Fragmentation and Allocation

Distributed database design has been dealt with in terms of fragmentation of data and allocation of these fragments. Figure 7-1 shows the various alternatives with which these problems can be attacked. For simplicity we do not show replication in this picture. The arm C in Figure 7-1 shows our proposed approach in data fragmentation and allocation. We will explain all the alternatives except our approach.

A: This approach expects that users specify fragmentation and allocation requirement together. Ceri et al. [Cer 83b] assumed that users are able to specify such candidates and derived horizontal partitioning schemes. The resulting design contains partitioning and candidates based on a 0-1 integer programming formulation. Replication was introduced after first solving the nonreplicated problem.

B: In this approach fragmentation and allocation are independently handled. Navathe et al. [Nava 84] applied this approach to vertical partitioning where fragments were designed first to achieve the best clustering of attributes based on intuitive empirical objective function. The allocation problem requires detailed cost information. They proposed an objective function based on costs of accessing multiple fragments, irrelevant attributes, etc, which can be weighted differently. Hâving designed the vertical fragments, they proposed separate allocation algorithms for both centralized and distributed environments.

C: This dissertation falls in this approach.

D: This approach corresponds to wide area networks where we would use allocation algorithms first to allocate the grid cells. Then based on the given allocation, local merging of fragments is performed to minimize overall transaction processing costs. Sreewastav [Sree 90] proposed an integrated system architecture for the initial design based on this approach.

E: This approach involves a more complex model of merging and allocation process where site dependent costs are brought into picture.

Among these alternatives we will explain the D approach in more detail in the following chapter because it is more related to our approach.

7.2 Cell Allocation Followed by Local Optimization

The grid cells produced after simultaneously applying the vertical and horizontal partitioning algorithms represent all possible partitioning of the relation based on the usage of the relation by the important transactions. The fragments of grid cells are the unit of allocation. The major difference from approach C is that in approach D grid cell-allocation is first performed without merging, and then grid cell merging is performed at each site. Basically there are two stages in this approach:

(1) Initial allocation of the fragments

In this stage each cell of the grid is allocated to sites. The objective of this stage is to optimize total processing cost. The allocation of fragments is not dependent of each other. The usage of fragments together by transactions is taken into account.

(2) Local fragment optimization

The local optimization of fragments refers to merging of cells allocated to a specific site, into larger fragments. The vertical and horizontal partitioning algorithms produce a grid of cells based on the attribute and predicate affinities. This stage generates all the possible fragments. The granularity of these fragments is very small; therefore a transaction may access a number of fragments, or a number of joins or unions are involved in transaction processing. Merging of these fragments, which are accessed together very frequently by transactions may result in savings due to an elimination of joins and unions.

Based on this approach, an integrated system architecture for the initial design of a distributed database was proposed in [Sree 90]. Sreewastav tried to adopt our mixed partitioning methodology to form a grid for each relation by simultaneously applying vertical and horizontal partitioning. The grid cells or fragments are then allocated nonredundantly using a greedy heuristic procedure. This initial allocation is followed by a replication procedure, which again is a greedy heuristic. A fragment is replicated at a site if the benefit of replication of the fragment at that site is positive and maximum among all possibilities. Finally, the fragments allocated to a site are further optimized with the objective of reducing processing costs attributed to join and union processing.

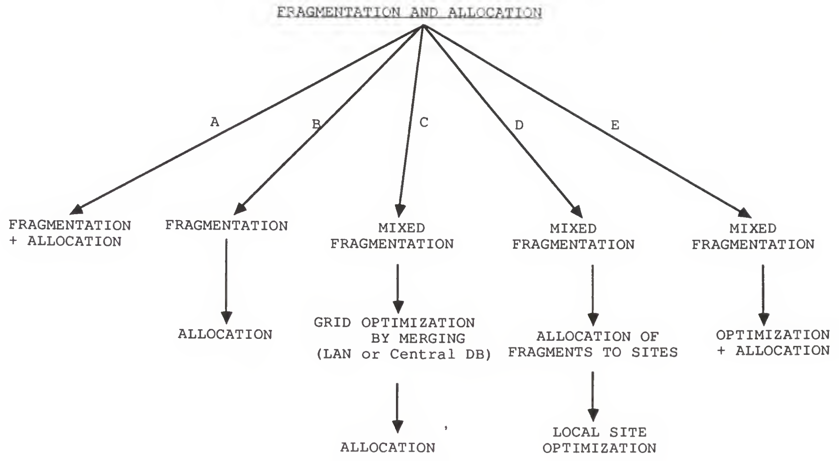

FRAGMENTATION AND ALLOCATION
A
B
C
D
E
FRAGMENTATION + ALLOCATION
FRAGMENTATION
ALLOCATION
MIXED FRAGMENTATION
GRID OPTIMIZATION BY MERGING (LAN or Central DB)
ALLOCATION
MIXED FRAGMENTATION
ALLOCATION OF FRAGMENTS TO SITES
LOCAL SITE OPTIMIZATION
MIXED FRAGMENTATION
OPTIMIZATION + ALLOCATION

Figure 7-1 Alternatives for fragmentation and allocation

CHAPTER 8
CONCLUSIONS AND FURTHER RESEARCH

The major issue of designing an efficient distributed database gives rise to problems of fragmentation and allocation of data. In this dissertation we have formally defined these problems and proposed a set of algorithms for partitioning and allocation. In our view, the major results and contributions of this work are the following.

This dissertation has presented a graph theoretical algorithm for vertical partitioning. In this algorithm all fragments are generated by one iteration in a time of $O(n^2)$ that is more efficient than previous approaches. Furthermore, it does not need an arbitrary objective function. We consider it very useful because it is applicable to any domain if an affinity matrix is provided. In fact, the horizontal partitioning algorithm proposed here is the result of applying this same algorithm to horizontal partitioning domain.

The mixed partitioning methodology allows the optimal partitioning of global relations in a distributed database by using a grid approach that incorporates horizontal and vertical fragmentations simultaneously. This is the first comprehensive treatment of mixed partitioning.

An allocation algorithm suited for mixed partitioning has been presented. We have specified four multi-fragment operations and defined allocation strategies based on the heuristic called the pseudoallocation technique by using a graphical method.

Further extension of this work will be in the direction of the following issues. The first issue is the development of an interactive design tool. This design tool will allow users to make fragmentation and allocation decisions for distributed databases using vertical partitioning, horizontal partitioning, mixed partitioning, and allocation.

The other issue is the investigation of a model for multiple copies of fragments in the allocation phase. Replication can be performed as a consequence of nonreplicated allocation by ascertaining whether there is cost benefit after replication. More work is needed to model the performance evaluation with respect to the response time of transactions and the effect of network topology.

Tying this work closely to actual environments where α, β, and J are actually computed for an actual system will be good.

REFERENCES

[Aper 88] Apers, P. M. G., "Data Allocation in Distributed Database Systems," ACM Trans. on Database Systems, Vol. 13, No. 3, September 1988, pp.263-304.

[Bras 88] Brassard, G., and Bratley, P., Algorithmics: Theory & Practice, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Cer 80] Ceri, S., Martella, G., and Pelagatti, G., "Optimal File Allocation for a Distributed Database on a Network of Minicomputers," Proc. International Conference on Databases, Aberdeen, Hayden, July 1980.

[Cer 83a] Ceri, S., and Navathe, S. B., "A Methodology to the Distribution Design of Databases," Proc. IEEE COMPCON Conference, San Francisco, CA., February 1983.

[Cer 83b] Ceri, S., Navathe, S. B., and Wiederhold, G., "Distribution Design of Logical Database Schemas," IEEE Trans. on Software Engineering, Vol. SE-9, No. 4, July 1983, pp.487-504.

[Ceri 82] Ceri, S., Negri, M., and Pelagatti, G., "Horizontal Data Partitioning in Database Design," Proc. ACM SIGMOD International Conference on Management of Data, Orlando, FL., June 1982.

[Ceri 84] Ceri, S., and Pelagatti, G., Distributed Databases: Principles and Systems, McGraw-Hill, New York, 1984.

[Ceri 87] Ceri, S., Pernici, B., and Wiederhold, G., "Distributed Database Design Methodologies," IEEE Proceedings, April 1987.

[Ceri 88] Ceri, S., Pernici, B., and Wiederhold, G., "Optimization Problems and Solution Methods in the Design of Data Distribution," Working paper, Stanford University, 1988.

[Cope 88] Copeland, G., Alexander, W., Boughter, E., and Keller, T., "Data Placement in Bubba," Proc. ACM SIGMOD International Conference on Management of Data, Chicago, IL., June 1988.

[Corn 87] Cornell, D. W., and Yu, P. S., "A vertical Partitioning Algorithm for Relational Databases," Proc. Third International Conference on Data Engineering, Los Angeles, CA., February 1987.

[Corn 88] Cornell, D. W., and Yu, P. S., "Site Assignment for Relations and Join Operations in the Distributed Transaction Processing Environment," Proc. Fourth International Conference on Data Engineering, Los Angeles, CA., February 1988.

[Corn 89] Cornell, D. W., and Yu, P. S., "On Optimal Site Assignment for Relations in the Distributed Data Environment," IEEE Trans. on Software Engineering, Vol. 15, No. 8, August 1989, pp.1004-1009.

[DeW 86] DeWitt, D. J., Gerber, R. H., Graefe, G., Heytens, M. L., Kumar, K. B., and Muralikrishna, M., "GAMMA - A high Performance Dataflow Database Machine," Proc. Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, August 1986.

[Dowd 82] Dowdy, L. W., and Foster, D. V., "Comparative Models of the File Assignment Problem," Computing Surveys, Vol. 14, No. 2, June 1982, pp.287-313.

[Elma 89] Elmasri, R., and Navathe, S. B., Fundamentals of Database Systems, Benjamin/Cummings Publishing, Redwood City, California, 1989.

[Gare 79] Garey, M. R., and Johnson, D. S., Computers and Intractability : A Guide to the Theory Of NP-Completeness, Freeman, New York, 1979.

[Hamm 79] Hammer, M., and Niamir, B., "A Heuristic Approach to Attribute Partitioning," Proc. ACM SIGMOD International Conference on Management of Data, Boston, MA., May 1979.

[Hoff 75] Hoffer, J. A., and Severance, D. G., "The Use of Cluster Analysis in Physical Database Design," Proc. First International Conference on Very Large Data Bases, Framingham, MA., September 1975.

[McCo 72] McCormick, W. T., Schweitzer, P. J., and White, T. W., "Problem Decomposition and Data Reorganization by a Clustering Technique," Operations Research, Vol. 20, No. 5, September 1972, pp.993-1009.

[Nava 84] Navathe, S. B., Ceri, S., Wiederhold, G., and Dou, J., "Vertical Partitioning Algorithms for Database Design," ACM Trans. on Database Systems, Vol. 9, No. 4, December 1984, pp.680-710.

[Sacc 85] Sacca, D., and Wiederhold, G., "Database Partitioning in a Cluster of Processors," ACM Trans. on Database Systems, Vol. 10, No. 1, March 1985, pp.29-56.

[Sree 90] Sreewastav, K., "A Distributed Database Initial Design System," Master's thesis, University of Florida, 1990.

[Swam 89] Swami, A., "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques," ACM SIGMOD International Conference on Management of Data, Portland, OR., May 1989.

[Wah 84] Wah, B. W., "File Placement on Distributed Computer Systems," IEEE Computer, Vol. 17, No. 1, January 1984, pp.23-33.

[Wied 87] Wiederhold, G., File Organization for Database Design, McGraw-Hill, New York, 1987.

[Yu 85] Yu, C. T., Suen, C., Lam, K., and Siu, M. K., "Adaptive Record Clustering," ACM Trans. on Database Systems, Vol. 10, No. 2, June 1985, pp.180-204.

BIOGRAPHICAL SKETCH

Minyoung Ra is from Korea. He graduated from Korea Military Academy, Seoul, Korea, in 1978. He received the bachelor's degree and master's degree in computer science at Seoul National University, Seoul, Korea, in 1983 and 1986, respectively. Before starting his Ph.D studies he worked as a faculty member at Korea Military Academy.

Since 1987, he has been working towards the doctoral degree in computer and information sciences at the University of Florida. His research interests are in the area of database management systems, distributed database design, federated information bases and semantic data modeling.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Shamkant B. Navathe, Chair
Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Ravi Varadarajan, Cochair
Assistant Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Sharma Chakravarthy
Associate Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Jose C. Principe
Associate Professor of Electrical Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate school and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December 1990

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School